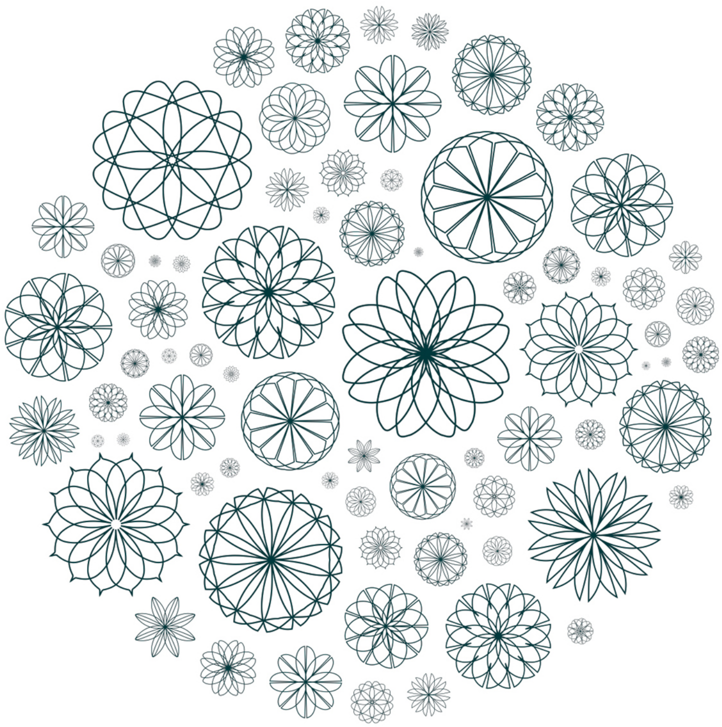
Bleeding Edge Press
Publishing at the speed of technology

TURING 图灵程序设计丛书

五位开发高手，三个企业应用示例，真实再现**React Native**开发场景
一次性开发跨平台应用、原生体验，开发**效率高**，满足前端开发**快速迭代**需求



Deconstructing React Native Apps

# React Native
# 应用开发实例解析

[澳] Alexander McLeod [斯洛文尼亚] Pavlo Aksonov
[印] Arjun Komath [美] Atticus White [美] Isaac Madwed 著

林昊 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

# 数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

## 作者

### Alexander McLeod

销售人员在线训练平台Myagi的CTO。

### Pavlo Aksonov

经验丰富的软件开发人员，活跃的开源贡献者，有超过15年的Web和移动端开发经验。

### Arjun Komath

精通多门语言的程序员，活跃的开源贡献者，用React Native开发了Product Hunt的开源Android客户端Feline。

### Atticus White

就职于波士顿的Robin Powered公司，精通React Native、Angular以及NodeJS开发。

### Isaac Madwed

全栈工程师，就职于Fixt。

## 译者

### 林 昊

毕业于华中科技大学，现就职于网龙工程院前端团队，热衷技术翻译，喜欢探索现代Web技术，对大前端的发展有浓厚兴趣。

图灵程序设计丛书

Deconstructing React Native Apps

# React Native 应用开发实例解析

[澳] Alexander McLeod [斯洛文尼亚] Pavlo Aksonov
[印] Arjun Komath [美] Atticus White [美] Isaac Madwed 著

林昊 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

React Native应用开发实例解析 / (澳) 亚历山大•
麦克劳德 (Alexander McLeod) 等著 ; 林昊译. -- 北京:
人民邮电出版社, 2017.9
（图灵程序设计丛书）
ISBN 978-7-115-46714-0

Ⅰ. ①R… Ⅱ. ①亚… ②林… Ⅲ. ①移动终端—应用
程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2017)第203246号

## 内 容 提 要

使用 React Native 可以轻松开发跨平台应用，并且无需等待 Apple、Google 或者 Amazon 的审核过程，就可以为自己的应用发布更新。本书主要从功能扩展和实际应用方面讲解 React Native，带领读者全面了解 React Native 的 API 和组件，并且阅读本书无需 React 开发背景。本书共五章，前两章介绍 React Native 的历史发展和基础知识，包括原生组件和第三方库；余下三章则分别介绍三个企业应用——Myagi、TinyRobot 和 Fixt，探讨了当今业界使用 React Native 的方式，以及生产环境下需要注意的问题和相应对策。

本书适合客户端开发人员、前端开发人员，以及所有对 React Native 感兴趣的程序员。

◆ 著　　[澳] Alexander McLeod
[斯洛文尼亚] Pavlo Aksonov　[印] Arjun Komath
[美] Atticus White　[美] Isaac Madwed
译　　林 昊
责任编辑　朱 巍
执行编辑　温 雪　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编 100164　电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：10
字数：237千字　　2017年 9 月第 1 版
印数：1－3 000册　　2017年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-5338号

定价：45.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京东工商广登字 20170147 号**

# 前　　言

## 什么是React Native？

你可能已经听说过React，这个由Facebook开发的框架已经流行多时，如今成为了现代Web开发的标准。React使得开发者可以编写和构建声明式组件，清晰地理解应用架构。React不会给开发者的其他技术栈造成冲突，可以与任意后端技术甚至其他前端技术搭配使用。

你可能在想："哇，这听起来很不错，但用在移动端会如何呢？我是否也能用React来写移动应用？"

实际上，有两种用React开发移动端的方式。React本身可以在移动Web上运行，这就意味着所有标准React元素都可用。然而这样做本质上还是开发Web应用，所有基于Web搭建的应用所面临的性能和权限问题，React应用同样会遇到。

如果可以在移动端用React进行原生开发，那么……

幸运的是，真的可以这样做！如果你打算开发原生移动应用，用React Native吧。React Native把React的模式与范例带到了原生移动开发中。它的思想与React相似，不对现有的应用架构和技术栈造成冲突。开发者可以把React Native与几乎所有的技术进行组合，搭建出满足各种架构模式的应用。

尽管两者非常相似，React和React Native之间依然有很多明显的差别。首先，React Native自带的基础组件库与React的不同。React Native的开发过程不需要编写`div`和`span`标签，而要使用视图以及文本组件，并且有权限调用诸如地理位置、通知推送、加速器数据、设备振动等大量原生工具。相比在移动浏览器中开发React应用，React Native赋予了开发者更多可能。

这时你可能会觉得，React Native与Apache的Cordova类似。诚然，两者的思想非常相似，都共用Android和iOS两个平台的代码库。然而，Cordova运行在WebView中，通过调用API获得原生级别的功能。React Native组件会被渲染成原生部件，可以为移动应用提供真正的原生体验，而Cordova应用在遇到滚动这样高强度的UI交互场景时，可能会发生崩溃——这就是React Native的威力所在。开发者可以受益于React声明式UI的编程风格；同时，所维护的用户界面提供的极速体验，能够与任何原生移动应用相媲美。

React Native的社区活跃且增长迅速，并且已经被Facebook等许多公司用于生产环境。本书将致力于帮助你理解React Native，以便弄清React Native是否适用于自己的应用。

## 我应该使用React Native吗？

你是否属于小型团队，并且你的团队想要为iOS和Android两个市场开发应用？

比起为iOS和Android两个平台开发各自的应用，React Native可以让你一次性开发跨平台应用，这样更省时省力。你可以更快地发布应用，从而将更多的时间和精力集中于用户体验，无需操心平台的差异性。

你是否正在使用NodeJS和其他JavaScript前端技术？

React Native由纯粹传统的JavaScript和JSX语法编写。开发React Native能让你在不同的应用环境中切换自如，无需因为编程语言的不同而改变开发环境。你还可以从npm以及其他资源仓库获取第三方JavaScript库用于应用开发。

你是否希望，无需等待Apple、Google或者Amazon的审核过程，就可以为自己的应用发布更新？

微软的CodePush服务集成了React Native的支持。用React Native进行开发，可以直接把JavaScript更新包部署到应用上，无需等待任何第三方服务的审核。比起正式发布的流程，你可以更快地修复bug和新增特性，并提供更广泛的兼容性。

你是否希望，能够维护更小的代码库，以便更清楚地构思、更快地发布应用？

React Native共用了iOS和Android的代码库。利用React Native，再小的团队也能够做更多的事情。除此之外，Web开发者可以立刻加入移动应用的开发，进而了解移动端的原生环境。

值得使用React Native的理由远不止上述这些。要弄清React Native是否适合你，唯一的方式就是了解它能够给你带来什么。本书将会带你过一遍React Native的代码库，并展示真实的应用是如何开发出来的。

## 阅读前提

本书假定你至少了解基础的JavaScript知识，并且熟悉ES2016语法。不过，即使不熟悉这些，大多数人在阅读本书的过程中也能学会。

阅读本书不需要有React的开发背景。React Native和React不同但存在交集，书中将会详细描述它们相关的部分。

## 本书内容

本书将指导你如何开始编写React Native应用，书中通过几个企业应用的实例，让你对当今业界如何使用React Native有一些认识。

第1章简要概述移动应用开发的现状，并介绍React和React Native的诞生。第1章将介绍JSX语法以及在React Native应用运行的过程中发生了什么，然后介绍一些React Native组件以及它们的生命周期和能力，此外还有部署过程与原生模块的使用。

第2章涉及原生组件的方方面面：用JavaScript、Java和Objective-C创建自定义原生组件；从组件、常量、事件中进行异步调用；链接第三方库。

第3章介绍Myagi应用。Myagi为零售销售人员提供训练平台。你在了解Myagi的过程中会接触到Marty.js、深度链接以及环境配置。本章将带领你实现一个自定义的构建脚本，以及学习如何在iOS、Android、Web应用之间共享代码。最后介绍维护无bug移动应用至关重要的一环——测试与质量保证，同时还将提到CodePush服务。

第4章介绍基于位置的移动聊天应用TinyRobot。你将学到用Flow进行静态类型检查，接着学习Flux、Redux、MobX以及它们的异同点，还将学习依赖注入、持久化以及应用状态管理。由于React Native是无结构化的，本章会带你了解一些设计模式，以及如何从UI中分离业务逻辑，还有如何实现UI测试。

第5章介绍Fixt，它同时为普通消费者和企业客户提供手机维修贵宾服务。通过讲解一系列基于React Native的解决方案，本章将指导你利用React Native实现特定的用途，并学习Fixt提供的React Native设备参数包。你还将了解到Fixt用React Native实现的Digits（Twitter的认证系统解决方案）。本书最后会给出进一步学习React Native的建议，并告诉你遇到难题时去何处寻求帮助。

## 代码示例

全书包含大量的代码示例，书中的JavaScript代码示例用ES2016语法编写，必要之处还附有注释。

## 关于作者

Isaac Madwed是一名全栈工程师，就职于Fixt，Fixt是一家提供手机维修贵宾服务的国际化公司，总部位于美国马里兰州的巴尔的摩市。他平时热衷于使用机器学习来控制艺术创作过程，并且喜欢整洁、模块化、声明式的编程方式。想要了解更多有关Fixt的信息，请访问网站www.fixt.co。想要欣赏Isaac的艺术作品，请访问网站www.imadwed.com。

Pavlo Aksonov是一名就职于Hippware的React Native UI软件开发人员。他是一名活跃的开源

贡献者，开发了React Native路由库Flux以及许多其他组件。Pavlo关注软件架构设计，并有超过15年的Web和移动端开发经验。你可以通过Twitter（@AksonovP）与他联系。工作之余，Pavlo喜欢打乒乓球。

Alexander McLeod是销售人员在线训练平台Myagi的CTO。加入Myagi之前，他取得了墨尔本大学的计算和软件系统学位，并曾在多家创业公司任职CTO。Alex大部分时间都在用JavaScript和Python编写Web和移动应用。工作之余，他喜欢研究VR项目、创作音乐、打篮球以及滑雪。你可以通过Twitter（@amcleodio）与他联系。

Arjun Komath来自印度，他不仅拥有计算机工程本科学历，还是一名精通多门语言的程序员。过去两年内，他一直与Web和移动端技术打交道。同样，他也是一名活跃的开源贡献者。他用React Native开发了Product Hunt的开源Android客户端Feline。本书第1章由Arjun执笔，你如果有任何疑问，可以通过Twitter（@arjunz）请教他。

Atticus White就职于波士顿的Robin Powered公司，是一名React Native、Angular以及NodeJS开发人员。该公司的产品Robin致力于让预订会议室更加简便，可以将Robin看成办公室版的OpenTable。他业余时间喜欢在实验室里研究开源项目并探索JavaScript的世界。想要了解更多有关Robin Powered的信息，请访问网站www.robinpowered.com。想要了解Atticus的更多信息，请访问网站atticuswhite.com或者通过Twitter（@atticoos）与他联系。

## 电子书

扫描如下二维码，即可购买本书电子版。

# 目　　录

# 第 1 章 用JavaScript开发移动应用

移动应用程序（简称移动应用）是专门为智能手机和平板电脑这样的小型移动设备开发的软件应用，它们不用于台式计算机或笔记本电脑。iOS的App Store、Android的Play Store等应用商店拥有数百万的移动应用，截至2015年6月，仅App Store上移动应用的下载量就超过了1000亿。这些惊人的数字表明应用市场一直在持续扩张和发展，这种情况也使得应用开发领域对软件开发者产生了特殊的吸引力。开发移动应用时，有多种技术可供选择。对于原生应用，最普遍的做法就是为特定的平台做原生的开发：比如Apple的iOS、Google的Android，以及Windows Phone等平台。除此以外，我们还可以开发混合应用，以及只能在移动浏览器中运行的简单Web应用。本章将对这些技术进行比较，帮助你理解为什么应该考虑React Native这个可以用JavaScript和React开发原生移动应用的框架。

开发移动应用没有开发Web应用那么简单。我们要面对好几个独立的平台（iOS、Android等）。这些平台的编程语言、原生组件和应用架构等都不一样。由于这些差异，即使是相当有经验的iOS开发者，可能也无法开发Android应用，因为Android的生态系统对于他而言完全是一个新的领域。

如果你是iOS或Android原生应用的开发者，在开发过程中很可能会遇到以下几种情况。

- **编译应用**：即使每次的改动都非常小，也需要重新打包整个应用，才能在模拟器或者真实设备上看到结果，这个过程严重拖慢了整个开发进度。
- **原生并不简单**：就算定义视图或布局这么简单的操作，也需要大量的代码。
- **一次只能给一个平台开发**：Android应用无法在iOS上运行，反之亦然。因此，为了给多个移动平台开发应用，你需要掌握不同的技术栈和工具集，而这一切只是为了写出一模一样的应用。
- **更新缓慢**：想象一下这种场景，你在生产环境中发现了一个bug，尽管经过调试并找到了解决方法，你也没办法将补丁魔法般地推送给下载过应用的用户。你不得不煞费苦心地打包应用、签名，最后把更新提交给应用商店。更新包上架到应用商店之前，所有用户会一直受bug困扰，由于应用市场政策的制约，我们对这种状况无能为力。

以上这些都表明，产品开发过程不但缓慢，而且代价昂贵。但原生并没有坏处，从性能角度来看，原生应用是最好的选择，它们的UI风格更统一，并且视觉和使用体验与整个平台融为一体。

简而言之，原生应用非常棒，但要付出一定代价。

为了解决上述问题，业界已经有了多种尝试，最流行的方式便是用HTML、CSS和JavaScript等Web技术开发移动应用。所有移动平台都有浏览器，因此可以运行任何Web应用。这种尝试很不错，但并非最佳解决方案。这些所谓的“混合”应用开发起来更简单，可以多平台运行并且不需要学习Objective-C或Java。开发者的美梦成真，但这样的应用对用户不友好。这些应用不好用，至少大部分都不好用！与原生应用相比，混合应用运行起来很慢，用户界面以及用户体验都很糟糕，而且在用户体验方面与原生应用完全没有可比性。

这时就轮到救星React Native登场了。它在两个领域都是最好的选择。在接下来的章节中，你将会看到，React Native把Web开发和原生开发完美地结合到一起。在开始编写React Native应用的壮丽旅程之前，先来了解一些构成React Native的基础概念，搞清楚Facebook如何以及为何要开发React Native。

## 1.1 过去

在React Native之前，有Cordova（前身为PhoneGap）和Ionic这样的跨平台应用开发框架，这些框架可以渲染JavaScript、HTML和CSS所编写的WebView。这些应用没有权限调用平台的特定组件，而是用HTML、CSS和JavaScript模拟这些组件。以Android平台的导航抽屉为例，它有很多基于Web的实现，但没有一个能达到完美的原生体验（如酷炫的汉堡包菜单翻转动画http://i.stack.imgur.com/tErny.gif），总是存在某些缺陷。用户体验以及应用的整体质感不可能像那些为宿主平台原生编写的应用一样优秀。从性能角度看，原生应用可以多线程运行，而Web平台的一切只能在主线程上运行。原生应用对手势的处理更加完善。这样的对比不胜枚举。

## 1.2 现状

React Native起源于2013年夏天的一次黑客马拉松项目。下面是引自Facebook的一段话。

> React Native的思想便是，我们能够把Web开发中各种深受开发者喜爱的东西带到移动开发领域，比如快速迭代、用一支团队开发整条产品线。这就意味着我们能做得更快。

在React Native诞生的早期阶段，用它开发的首批项目之一是个新闻订阅应用的原型。2014年7月，Facebook工程师想要为广告管理开发一个独立版本的iOS应用，唯一的难题在于广告团队里没有一个工程师有iOS开发经验。他们发现React Native是当时的完美选择，尽管那时它还处于原型阶段。接下来的数月，Ads Manager产品团队和React Native团队合作开发了第一个完全由React Native驱动的应用，它的体验与原生iOS应用一样棒。

这个iOS应用的成功使他们突破了正在构建的产品的极限，接着他们在伦敦创立了React Native Android团队。他们想要iOS版Ads Manager的代码在Android上运行，而且真的做到了。2015

年1月，Android版Ads Manager的可用原型开发完成，尽管这个应用从性能角度来看还不够好，少了很多特性，但它证明了未来有React Native的用武之地。

2015年1月的React.js大会，React Native的首个公开预览版亮相（https://www.youtube.com/watch?v=KVZ-P-ZI6W4）。2015年3月的F8开发者大会，Facebook把它开源提供给所有人。

2015年2月Facebook Ads Manager的iOS版发布，接着2015年6月他们又发布了Android版的Ads Manager。令人吃惊的是，这两个应用共享了约85%的源代码。

如今，React Native迅速被人们接纳，这一点业界有目共睹，它已成为最好的跨平台原生移动应用开发框架之一。2016年F8大会期间，微软和Facebook共同宣布，React Native新增了对通用Windows平台（UWP）的支持（https://blogs.windows.com/buildingapps/2016/04/13/react-native-on-the-universal-windows-platform/#O25TpvFJPKjSS67Z.99），这意味着你可以用React Native为Windows Phone、个人计算机甚至Xbox和HoloLens开发应用。

据我们所知，React Native并不是首个用Web技术开发移动应用的框架，但什么原因让它从现有的框架中脱颖而出并做得更好呢？为了理解这些问题，需要对React进行深入的探究。

## 1.3 React 的起源

React，这个用于构建用户界面的JavaScript库，就是React Native的核心。为了理解React，先要熟悉几个概念。第一个概念，声明式编程范式（范式就是计算机程序架构与组件的构建风格），用这种范式表达计算逻辑时不需要描述控制流程。简单地说，声明式编程就是你编写代码描述想要做什么，而不是怎么做。

第二个概念，异步，大多数JavaScript开发者已经很熟悉。同步是指“按顺序执行一段代码”，代码语句一行接一行地执行。这意味着每行代码都要等待前一行执行完成。异步代码让代码语句从主程序流程中脱离，主程序代码在异步调用之后立即继续执行，而无需等待异步代码完成。

> React具有**声明式**、**异步**、**响应式**的特性，使代码**可预测**，并让我们更有**把握**进行快速迭代。

### 1.3.1 为什么选择 React

HTML编写的Web应用中有文档对象模型DOM。DOM通过对象的形式来展现结构化文档。对于Web开发者来说，文档即HTML代码，DOM又称作HTML DOM，HTML的元素在DOM中叫节点。Web浏览器负责处理DOM的具体实现，并提供API接口以便对DOM进行遍历和修改。这样我们就能用JavaScript和CSS与DOM交互，比如查找节点并修改内容、移除节点、插入新节点。无论何时想要动态改变网页内容，只要通过API接口修改DOM即可（如今的DOM API几乎实现了跨平台和跨浏览器的兼容性）。

不过，关键问题在于DOM没有对动态创建UI进行优化。尽管可以用JavaScript和jQuery库操作DOM，但大量的操作就会引发性能问题。那么React是如何解决这个难题的呢？

React的开发者采取了虚拟DOM的做法，虚拟DOM更加轻量，对真实DOM进行了抽象化，而且独立于特定浏览器的具体实现。每当触发需要改变DOM的事件时，React会创建一个新的虚拟DOM树，并将其与已有的树进行对比，计算出最少的DOM变化集合，把它们放入队列再全部批量执行，接着重新渲染视图。这种做法没有把重负荷操作全部加在真实DOM 上，因此比直接操作DOM快了很多。React的这种行为没有采用脏数据检查（dirty checking，持续检测模型变动），而是利用观察者模型进行变动检测，通过差分算法（diffing algorithm，http://calendar.perfplanet.com/2013/diff/）判断最少的DOM操作，因此很有效率。

> React为可变命令式DOM接口编程提供了**声明式**封装。声明式的编程方法只需**描述程序应该达成什么目的**，而不用关心程序应该怎么运行。然而命令式编程需要逐步编写程序，将输入值计算成期望的输出。

- **开发简单，声明式编程**：只需告诉React，你希望应用长成什么样。按照设计稿编写声明式视图，定义应用的状态。React会根据应用状态，仅更新、渲染对应的组件，非常高效。这让代码的编写和维护变得非常容易，同时更具可预测性，更容易调试。
- **组件化开发**：Facebook宣称，“用了React，你只需要开发组件”。开发一整套组件，再把它们拼装成应用。

现在我们知道了React是什么，接下来简要了解一下这些魔法背后的原理。

### 1.3.2　React 的工作原理

编写React应用的首要任务便是开发组件，组件属于视图部分，同时也定义了应用状态，而数据层内容取决于当前渲染的视图。

- **将UI拆解成组件**：组件驱动开发，是指将代码拆分成组件，理论上组件只负责一件事。有个很简单的技巧，称为单一职责原则（the single responsibility principle），指一个组件理论上只应该做一件事。如果组件需要进一步扩展，就应该将它拆解成更小的子组件。这样可以让代码更容易理解、维护和测试。
- **交互式UI**：要使UI具有交互性，你需要触发UI背后的数据模型的改变。React中通过状态很容易做到这点。每个组件拥有内部状态、逻辑、事件处理器（如点击按钮和改变表单输入），也可以包含行内样式。一旦状态发生了改变，React就重新渲染视图。

需要注意的是，React有两种类型的数据“模型”：属性（props，英文properties的简写）和状态（state）。弄清两者的区别很重要。简而言之，如果组件有时候需要修改自身的某个特性，那么这个特性应该归类于组件状态的一部分，除此以外便是组件的属性。属性可以比作组件的静态数据，而状态是动态的，不过两者都可以触发重新渲染。

现在我们知道了React的工作原理，接着继续了解一下同样的概念如何打造出React Native。

## 1.4　为什么选择 React Native

React最神奇的地方在于，它被设计成能对任意的命令式视图系统进行封装，不仅限于DOM。因此Facebook的工程师某天就想到，为什么不用React来封装真正的原生移动UI呢？React Native就这么诞生了！

正如React用虚拟DOM产生的魔法一样，React Native通过原生宿主平台的API也实现了一样的效果。React Native应用借助宿主平台上Objective-C语言（iOS平台）或Java语言（Android平台）的UI库，渲染真正的原生UI组件，不仅限于WebView，这就解释了为何React Native能给应用带来更强的性能、更贴近原生的视觉感受以及使用体验。

> React Native允许开发者通过JavaScript函数的代理，直接调用原生模块。

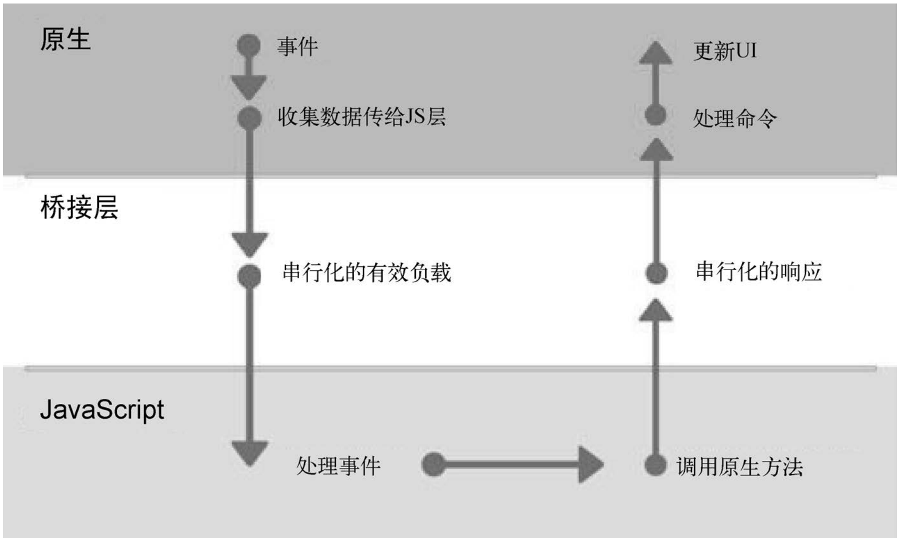
从性能角度上分析，React Native把所有应用代码和业务逻辑从主线程转移到后台线程运行。它可以批量处理要原生执行的请求，等控制权转让给主线程时再异步执行。React Native会分析你的UI，将最少的数据传给主线程（又称UI线程）以便用原生组件进行渲染。

> 使用React Native，你会得到原生的用户体验以及Web的开发体验。

- **简单易学**：如果你曾经开发过移动端，你可能会惊讶于React Native如此简单易用。
- **快速迭代**：不用等待应用构建，只要按下刷新快捷键，所有改动会立刻在应用中呈现。
- **智能调试**：与React一样，React Native也会在报错时抛出简明扼要的描述信息。
- **原生模块**：React Native的设计目的就在于，让你能够编写真正的原生代码，并在自定义原生模块的帮助下获得平台提供的完整能力。
- **一次性学习，全平台开发**：同一支工程师团队可以为任何平台开发应用，不需要为每个平台学习不同的基础技术。

## 1.5　React Native 的工作原理

原生代码与JavaScript代码通过桥接层进行交互，这是一个异步的批量串行处理过程。桥接层介于原生层和JavaScript代码之间，正如它的名称一样，它的作用很像桥（bridge）。用户输入、计时器、网络请求和响应等事件注册在原生代码中。React Native在原生层收集事件产生的数据，串行处理后通过桥接层传给JavaScript层。JavaScript层拿到数据后处理并生成一系列指令。这些指令由整型、字符串等数据类型构成，同样经过批量串行处理后传回原生层。桥接层的原生端决定哪个原生模块负责处理传回的指令并调用相应的方法，同时在需要的情况下更新UI。这种架构提升了React Native应用的性能，并且让React Native应用能以每秒60帧的速率运行。




React Native需要通过JavaScript执行环境运行JavaScript代码。在iOS和Android系统的模拟器以及设备上，React Native使用Safari的JavaScript引擎JavaScriptCore（http://trac.webkit.org/wiki/JavaScriptCore）。通过Chrome调试React Native时，JavaScript代码会在Chrome V8引擎内运行，并通过WebSocket与原生代码进行交互。React Native Windows应用的JavaScript运行环境是Chakra（微软Edge浏览器的JavaScript引擎https://github.com/Microsoft/ChakraCore），UWP（通用Windows平台）应用包不需要添加任何额外的二进制文件就可以使用Chakra引擎。

## 运行 React Native 应用时发生了什么

下面来看看React Native应用启动时发生了什么。启动应用时有以下三个任务并行完成。

- **加载JavaScript打包文件**，React Native的打包工具会像Webpack和Browserify一样把代码连同全部依赖打包成单个文件。
- 与此同时，React Native开始**加载原生模块**。一旦某个原生模块完成加载就在桥接层注册，桥接层确认该模块。此时整个应用便知道该模块已可用并能创建该模块的实例。
- **启动JavaScript虚拟机**，提供JavaScript代码的执行环境。

一旦原生模块和JavaScript执行环境准备就绪，应用就会加载JSON配置文件。配置文件中包含了模块数组、常量导出模块以及模块的方法。这个文件的重要之处在于，当你请求依赖的原生模块并调用方法时，JavaScript会读取它并在执行环境中创建对象。

下一步，执行JavaScript打包文件。创建Shadow视图（负责计算布局的独立线程称为shadow队列）来渲染应用的布局。Shadow队列把flexbox这样的属性转换成绝对位置和大小。与此同时，创建原生视图来渲染应用。最后结合两者将整个应用渲染到屏幕上。

现在来考虑一下应用中触发事件或者发生交互的情况。iOS系统的UIKit会识别触发的事件，并通过原生模块把事件分发给React（传给JavaScript层）。JavaScript捕获事件后会调用对应的事件处理器。如果事件处理器需要再次调用原生组件，将通过桥接层调用原生组件方法。

需要补充的是，iOS系统所有的原生模块有各自的线程池（GCD队列），而Android系统的模块共享同一个线程池，但两者的原生模块线程池都脱离于主线程存在。

以上对React Native工作原理的解释还处于很高的层级，在底层执行过程中实际上还有很多优化和动态的过程。

## 1.6　局限性

React Native在很多方面都十分神奇，但凡事总有缺憾。它仍然处于早期阶段，尚未发布稳定版，API也一直在变动。尽管Facebook在生产环境中使用它已经有一段时间，但如果要问它是否已经准备好用于生产环境，我们无法给出肯定的答复。根据我们使用React Native的经验，应用有时能运行得很棒，不过它确实会发生一些没人能弄清楚的问题。

React Native项目自2015年开源以来，就引起了业界的高度关注。

## 1.7　开发第一个 React Native 应用

我们将在本节学习如何用React Native开发一个简单的应用。首先来了解一些基础概念，熟悉了这些概念之后即可开始编程。

### 1.7.1　JSX——JavaScript 语法扩展

用React开发时，我们在render方法中用JSX语法代替JavaScript函数。JSX是看起来与XML类似的JavaScript语法扩展。它让编码更加方便，尤其是对那些Web技术开发者而言。与XML一样，JSX标签包括标签名（tag name）、特性（attribute）以及子元素（children）。两侧加有引号的特性值是字符串。除此之外，用花括号括住的值是封闭的JavaScript表达式。

JSX标签实际上调用了JavaScript函数，如下所示。

```
<div prop="someText">Children</div>;
```

上面的JSX标签编译成JavaScript后如下所示。

```
React.createElement("div", { prop: "someText" }, "Children");
```

### 1.7.2　状态和属性

1.3.2节探讨过状态和属性之间的区别。属性是“父组件向子组件传递数据的方式”。为子组

件指定属性并赋值后，子组件内就能通过this.props.propName的形式访问这些值。要记住子组件的属性由父组件提供，因此它们不能**直接**修改属性，从组件自身的视角来看，它的属性不可变。

状态，顾名思义，表示应用或组件的状态。与属性不同，状态是可变的和私有的，并且可以被组件自身修改。当状态发生改变时，React会自动重新渲染组件。要改变状态的值，需要调用组件内部的setState()方法。

举个例子，创建一个头部组件，命名为Header。为Header组件传递title属性并渲染。

```
<Header title="Hello World!" />
```

组件可以通过this.props.title来访问title属性。

```
class Header extends Component {
  render() {
    return (
      <View>
        <Text>{this.props.title}</Text>
      </View>
    );
  }
}
```

为了理解组件的状态，我们必须在应用内触发一个事件，比如用户输入或者网络请求。下面来尝试编写一个简单的计数器示例。

```
class Counter extends Component {

  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    setInterval(() => {
      this.setState({
        count: this.state.count + 1
      })
    }, 1000);
  }

  render() {
    return (
      <View>
        <Text>{this.state.count}</Text>
      </View>
    );
  }

}
```

在上面的示例中，我们创建了一个继承自React Component的Counter类，并在构造函数中初始化count状态，赋值为0。生命周期方法componentDidMount在render方法执行后被调用。在下一个话题中，我们会讨论更多的生命周期方法。在componentDidMount方法内调用setInterval函数，每隔一秒让count增加1。每当count的值递增时，React会调用render方法并重新渲染UI。因此当你运行应用时就会看到计数器逐秒递增，render方法一直对Text组件内部的值进行更新。

### 1.7.3　React 组件生命周期

下面来看一下组件的各个生命周期方法的职责以及调用时机。理解这些方法后，你就会明白应该调用哪个方法以及该方法合适的调用时机。

- componentWillMount：首次渲染之前调用一次。该方法在渲染之前被调用，因此当你想在组件挂载之前进行某些操作时，这个方法很有用。
- componentDidMount：首次渲染之后调用一次。该方法用来发起网络请求以及更新状态。通常用来集成其他JavaScript框架和异步函数。
- componentWillReceiveProps：该方法在属性发生变化时被调用，首次渲染过程不会触发该方法。
- shouldComponentUpdate：该方法可以用来判定下一步是否应该执行render方法，也就是决定组件是否应该更新。该方法的返回值默认为true。如果更新状态或属性后不想渲染组件，可以在该方法中返回false。
- componentWillUpdate：组件重新渲染之前调用。
- componentDidUpdate：组件重新渲染之后调用。
- componentWillUnmount：组件卸载之前调用。

### 1.7.4　样式

应用的UI负责与用户进行交互，因此它是至关重要的一部分。我们自然会对那些UI惊艳的应用一见倾心。React Native没有实现对CSS的支持，而是通过JavaScript来给应用增加样式。你可以使用StyleSheet对象声明样式。

```
var styles = StyleSheet.create({
  background: {
    backgroundColor: '#222222',
  },
  active: {
    borderWidth: 2,
    borderColor: '#00ff00',
  },
});
```

- StyleSheet.create构造方法不是必要的，但是它有一些显著的优势。它把参数值转换成

普通的数值，数值的引用指向一张内部数据表，这样能够保证参数值不可变并且不能直接访问。

- 把样式表写在文件的末尾，可以确保样式只会创建一次，避免应用在每次渲染的过程中都重复创建。

所有核心组件都接受一个样式属性，它们也接受一组样式。

```
<Text style={styles.active} />
<View style={[styles.active, styles.background]} />
```

数组形式的样式作用于组件的规则与`Object.assign`方法一样：如果存在重名的值，最右侧的数组元素拥有较高的优先级，同时`false`、`undefined`和`null`这样的假值会被忽略。

我们在React Native中用flexbox进行布局。flexbox为页面元素的排列提供了一种布局模式，它有助于编写可预测的UI布局，以便适配不同的屏幕尺寸和显示设备。

flexbox实质上有以下三个主要属性。

- `direction`可以将页面元素按行（水平排列）或者列（垂直排列）的方向进行布局。
- `justify-content`可以使用该属性根据布局的主轴方向（水平或垂直，取决于所设置的`direction`属性）对页面元素进行对齐。该属性有以下五种值：`flex-start`，所有弹性元素对齐到主轴的起始位置；`flex-end`，所有弹性元素对齐到主轴的结束位置；`center`，在主轴方向上让所有弹性元素居中；`space-between`，所有弹性元素在主轴方向上等间距排列（布局内除了弹性元素以外的剩余空间，在元素之间平均分配）；`space-around`，弹性元素的前后以及相邻元素之间都留有空间。
- `align-items`用来在副轴方向上对齐元素，副轴与主轴垂直。该属性有以下四种值：`flex-start`，所有弹性元素对齐到副轴的起始位置；`flex-end`，所有弹性元素对齐到副轴的结束位置；`center`，在副轴方向上让所有弹性元素居中；`stretch`，该属性会拉伸弹性元素的尺寸来填满副轴长度。

### 1.7.5　触摸事件的处理

React Native可使用抽象实现的组件`TouchableHighlight`和`TouchableOpacity`来处理视图上的触摸事件。这两个组件都用来包装视图，使得它们可以正确地响应点击事件，同时在用户进行触摸时提供视觉上的反馈。另一个组件`TouchableWithoutFeedback`的用途一样，只是不提供视觉反馈。

```
<TouchableHighlight onPress={this._onPressButton}>
  <Text>Press me</Text>
</TouchableHighlight>
```

在上面的代码示例中，当点击文本Press me时，`TouchableHighlight`组件将调用`_onPressButton`函数，我们可以在该函数中对触摸事件执行所需的操作。

### 1.7.6　网络

如今大多数的应用都需要与服务器（后端）进行交互，目的在于把数据保存到数据库中，对用户进行身份验证，启用通知推送等。在React Native中我们可以通过网络的腻子脚本库（polyfill）与服务器交互。有三种网络相关的方法：用fetch方法发起HTTP请求、用WebSocket进行全双工通信，以及XMLHttpRequest（XHR）方法。

最普遍使用的方法就是fetch，该方法用于与服务器API进行交互。我们来看一个简单的GET请求。

```
fetch('https://httpbin.org/get')
.then((response) => response.json())
.then((responseJson) => {
  console.log(responseJson);
})
.catch((error) => {
  // 处理错误信息
  console.warn(error);
});
```

上面的代码示例展示了一个简单的HTTP GET请求，先获取数据，再把文本解析成JSON，最后将响应值输出到控制台。需要注意的是，fetch方法返回一个promise，因此可以跟着调用then、catch以及done方法。

fetch方法也能用ES7语法的async和await特性来调用，如下所示。

```
class MyComponent extends React.Component {
  ...
  async getData() {
    try {
      let response = await fetch('https://httpbin.org/get');
      let responseJson = await response.json();
      return responseJson.users;
    } catch(error) {
      // 处理错误信息
      console.error(error);
    }
  }
  ...
}
```

### 1.7.7　深度链接

在开发应用的过程中，你可能会遇到使用开放授权协议OAuth进行用户身份验证的需求，在这种情况下应用可以代表你的用户完成某些操作。为了授权给用户，首先我们将被重定向到登录页面，在该页面中用户可以输入其登录信息（用户名和密码），同时可以对服务进行验证以代表用户自身为应用提供访问权限。该服务会返回授权令牌给应用，以后应用每次访问服务时都会附带这个令牌，以便服务端对请求进行验证。来看一个例子。

我们来给Product Hunt网站[1]开发一个Android客户端，希望应用提供登录功能，这样用户就可以通过应用访问Product Hunt的服务。首先，在Product Hunt的API控制台页面注册应用，得到客户端ID以及密钥（用于OAuth协议）。登录过程一开始，将用户重定向到身份验证页面的URL，比如https://api.xyz.com/v1/oauth，并在参数中附带客户端ID以及重定向URI。一旦用户对应用进行了验证，服务就会带领用户前往指定的重定向URI，并附上访问令牌。此时深度链接就派上用场了，在它的帮助下，我们可以让重定向URI直接打开应用，于是整个登录过程就完成了。

> 深度链接通过**统一资源定位符**（uniform resource identifier，URI）指向移动应用内部的具体位置，而不仅仅是简单地启动应用。

React Native使用`Linking`组件接受深度链接。我们在此将只讨论输入的应用链接，尽管`Linking`组件所提供的通用接口都能处理输入和输出的应用链接。

在Android和iOS系统中，我们都要改动原生代码来接受深度链接。正如下方Android应用的例子一样，我们按文档的指示，在manifest文件中添加intent filter过滤器。

```
<intent-filter android:label="@string/filter_title_viewgizmos">
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <!-- 接受以http://www.example.com/gizmos开头的URI -->
  <data android:scheme="http"
        android:host="www.example.com"
        android:pathPrefix="/gizmos" />
  <!-- 注意：pathPrefix属性值必须以/开头 -->
  <!-- 接受以example://gizmos开头的URI -->
  <data android:scheme="example"
        android:host="gizmos" />

</intent-filter>
```

如你所见，上面的示例展示了应用将会接受以http://www.example.com/gizmos和example://gizmos开头的URI。

iOS平台上需要编辑AppDelegate.m文件。

当应用通过注册的外部URL启动时，你可以使用`Linking`组件获取该URL并进行处理。

```
componentDidMount() {
  var url = Linking.getInitialURL().then((url) => {
    if (url) {
      console.log('Initial url is: ' + url);
    }
  }).catch(err => console.error('An error occurred', err));
}
```

这样就能处理任何输入的应用链接并执行相应的操作。

① 这是一个用户可以分享以及发现新产品的网站，网址为https://www.producthunt.com/。——译者注

## 1.7.8 动画

React Native内置的动画API可以创建流畅的动画，提供更佳的用户体验。React Native拥有两套动画系统：LayoutAnimation和Animated。LayoutAnimation作用于全局，对下一次渲染过程的所有变化应用动画。与此不同，Animated库有丰富的配置项，能应用在特定组件上。请看以下的例子。

```
class SampleApp extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      fadeAnim: new Animated.Value(0),
      fadeAnim2: new Animated.Value(0)
    };
  }

  componentDidMount() {
    Animated.timing(this.state.fadeAnim, {
      toValue: 1,
      duration: 1000
    }).start();
    Animated.timing(this.state.fadeAnim2, {
      toValue: 1,
      delay: 1000,
      duration: 1000
    }).start();
  }

  render() {
    return (
      <View style={styles.container}>
        <Animated.Text style={styles.welcome, {opacity: this.state.fadeAnim}}>
          Welcome to the React Native!
        </Animated.Text>
        <Animated.Text style={styles.welcome, {opacity: this.state.fadeAnim2}}>
          Im feeling lucky..
        </Animated.Text >
      </View>
    );
  }
}
```

在上面的示例中，我们在应用渲染完成后展示了文本的淡入动画。为了实现这种效果，使用了React Native的Animated API让不透明度从0变为1，这样就实现了淡入效果。

在构造函数中，初始化一个名为fadeAnim: new Animated.Value(0)的新状态，即一个从0开始缓动的新值Animated.Value。为了实现淡入效果，需要让fadeAnim值从0变为1，为此我们使用了Animated.timing。它与计时器类似，可以用来定义持续一定时间的动画。

```
Animated.timing(this.state.fadeAnim, {
  toValue: 1,
  duration: 1000
}).start();
```

将需要应用动画的值作为参数传入，比如本示例中的状态fadeAnim，同时在另一个参数内指定动画效果的最终值（toValue）以及具体的动画持续时间（duration）。上述示例中fadeAnim将在1000毫秒内从0变为1。最后，我们把它赋值给动画组件的样式。

```
<Animated.Text style={styles.welcome, {opacity: this.state.fadeAnim}}>
  Welcome to the React Native!
</Animated.Text>
```

如上所示，我们已将fadeAnim状态附加到Animated.Text的不透明度，因此文本就产生了淡入动画。Animated.timing还可以传入delay参数，用于控制动画在指定的延迟（单位为毫秒）后再开始执行。下面的代码展示了delay参数。

```
Animated.timing(this.state.fadeAnim2, {
  toValue: 1,
  delay: 1000,
  duration: 1000
}).start();
```

在1000毫秒之前，不透明度一直为0，当延迟阶段结束后，不透明度才开始从0变为1。

### 1.7.9　调试与热模块重载

利用Google Chrome浏览器可以很方便地调试React Native应用。要进行调试，打开iOS或Android平台Chrome应用内的开发者菜单，选择远程调试JS选项。Chrome浏览器会打开新的标签页，该页面会与打包器（打包器为应用提供JavaScript打包文件）建立WebSocket连接。一旦调试工具在Chrome内打开，设备会请求浏览器通过WebSocket消息加载应用脚本。此时Chrome会新建一个<script>标签来加载JavaScript打包文件，于是JavaScript代码就会在浏览器中运行。由于在Chrome浏览器上运行着与设备上完全一样的JavaScript代码，可以利用Chrome的所有调试工具来调试代码。

在代码中写下console.log('something')，内容就会被输出到Chrome的控制台上。在代码中插入debugger;声明，Chrome就会让代码执行停在对应的那一行，你可以查看此处的局部变量、全局变量，或者接着调试应用。你也可以使用console.warn()和console.error()，把应用内的警告和错误信息输出到Chrome中。

Facebook在版本0.22中引入了热模块重载（hot module reloading），该特性可以让应用在处于运行状态时，重新加载你所做的改动。Facebook对热重载的介绍如下。

> 热重载的概念就是指应用在执行环境下运行时，可以直接注入编辑过的文件。这样在调整UI时特别有用，因为任何应用状态都不会丢失。

### 1.7.10 应用监控

- **性能监控器**

性能监控器提供了一个面板，遮罩在应用界面的最上层，用于显示与性能相关的数据。第一项RAM表示为当前进程分配了多少内存。第二项JSC表示JavaScript核心堆内存的大小（只有当JSC在执行环境中可用时，这项数据才会显示）。第三项Views表示应用拥有的视图数量以及当前屏幕上显示的视图数量。第四项UI表示主线程帧率。最后一项JavaScript表示JavaScript线程帧率。依靠这些数据我们就可以弄清应用哪里存在性能瓶颈。

如果JavaScript线程的帧率出现了明显的下跌，就意味着JavaScript代码执行得很慢。为了深入了解代码出了什么问题，就要用到Systrace工具以及CPU分析器。

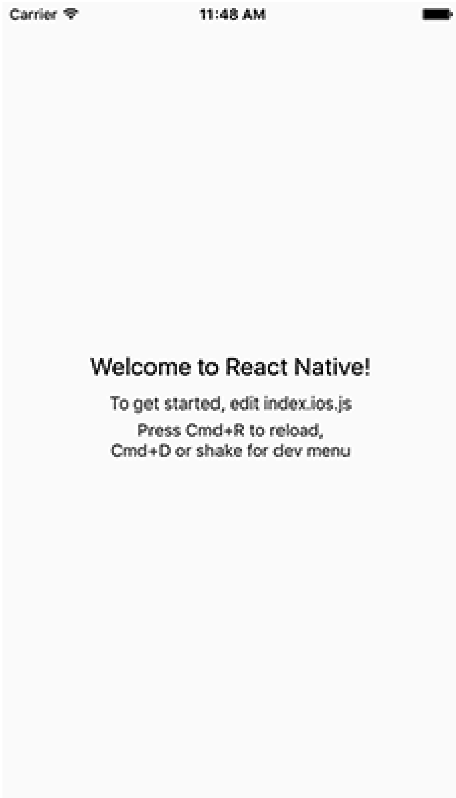
- **Systrace**

Systrace是以标记为基础的性能分析工具，这意味着要在应用内显式添加代码来获取性能信息。性能分析代码被开始/结束标记所包围，并高亮成彩色的语法格式。React Native默认会为应用以及自定义模块注入一些标记。同时React Native也提供API用于创建自定义标记。

## 1.8 开始动手

我们利用开源的电影数据库API开发一个简单的应用，用来搜索影片。在开始之前请确保安装了React Native，如果没有，运行`sudo npm install -g react-native-cli`命令进行安装。Node和npm是安装React Native的必备条件。

强烈建议通过brew（macOS的包管理器）安装watchman（用于监视文件变化）以及flow（用于JavaScript代码类型检查）。完成上述准备工作后，就可以运行`react-native init movies`命令初始化一个新的项目。这样会初始化一个新文件夹，包含了iOS和Android两个平台的原始项目文件以及JavaScript文件。运行`cd movies`打开项目目录，然后运行`react-native run-ios`，就能在iOS模拟器中预览应用。




来看一下React Native生成的JavaScript代码样例。

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
  Text,
  View
} from 'react-native';
```

示例一开始的`import`声明导入了React以及编写组件时所要继承的`Component`类。接着显式指定要使用哪些组件，比如`Text`和`View`，还有我们所需的`AppRegistry`和`StyleSheet`库函数。因此无论何时需要什么React Native组件，只要通过`import`声明导入即可。

```
class movies extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
```

```
          <Text style={styles.instructions}>
            To get started, edit index.ios.js
          </Text>
          <Text style={styles.instructions}>
            Press Cmd+R to reload,{'\n'}
            Cmd+D or shake for dev menu
          </Text>
        </View>
      );
    }
  }

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: '#F5FCFF',
    },
    welcome: {
      fontSize: 20,
      textAlign: 'center',
      margin: 10,
    },
    instructions: {
      textAlign: 'center',
      color: '#333333',
      marginBottom: 5,
    },
  });
```

接着创建一个movies类，继承自React Component。movies拥有render方法，返回一个由文本构成的布局。接着用StyleSheet新建一个样式表，指定render方法中的组件需要使用的样式对象。

```
AppRegistry.registerComponent('movies', () => movies);
```

最后要用到AppRegistry，为运行React Native应用提供JS代码入口。AppRegistry把movies组件暴露给应用的原生系统，使得原生系统能够为应用加载代码，让应用真正运行起来。

我们已经剖析了示例应用的代码，下面就开始编写自己的应用吧。

## 1.9　第一步：编写用户界面

我们的应用一开始只有一条路由，指向搜索页面，可称之为主路由。接下来用核心的Navigator组件来搭建路由系统。

React Native中的导航基于栈，也就是说完整的路由历史存储为数组形式。当我们想要导航至新的路由时，该路由就会被推入数组，并且在返回时再把它从数组中弹出。使用了Navigator组件后，index.ios.js文件的内容如下所示。

```
import React, { Component } from 'react';
import {
  AppRegistry,
  Navigator
} from 'react-native';

import Main from './components/main'

class movies extends Component {

  renderScene(route, navigator) {
    if (route.id === 'MAIN')
      return <Main navigator={navigator} />;
  }

  render() {
    return (
      <Navigator style={{ flex: 1 }}
        initialRoute={{ id: 'MAIN', title: 'Search Movies' }}
        renderScene={this.renderScene}
      />
    );
  }
}

AppRegistry.registerComponent('movies', () => movies);
```

render方法中的Navigator组件拥有initialRoute属性，这就是应用的主路由。同时代码中还有一个renderScene方法，它会根据指定的路由渲染对应的界面。在renderScene方法中我们判断路由id是否为MAIN，如果该条件满足就返回Main组件。

Main组件放在名为components的独立目录下，代码文件的内容如下所示。

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Text,
  View
} from 'react-native';

export default class main extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
```

```
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  }
});
```

到目前为止，Main组件仅仅在屏幕中间显示了一条文字信息：“Welcome to React Native!”在模拟器中刷新应用，就能看到如下所示的界面。




我们要在Main组件中添加两个组件，`TextInput`组件用于输入搜索关键词，`ListView`组件用于渲染搜索结果列表。实践中最好是始终将`ListView`或者`ListView`的子项拆分成独立的组件，但是为了简单起见，我们暂时把所有代码都写在一个文件中。

```
export default class main extends Component {

  constructor(props) {
    super(props);
    var ds = new ListView.DataSource({ rowHasChanged: (r1, r2) => r1 !== r2 });
    this.state = {
      dataSource: ds.cloneWithRows(['first item', 'second item']),
    }
  }

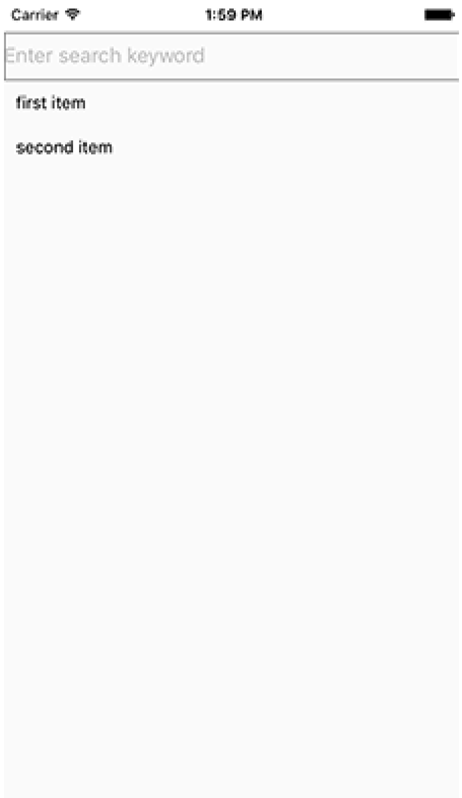
  renderRow(row) {
    return (
      <View style={styles.listItem}>
        <Text>{row}</Text>
      </View>
    )
  }

  render() {
    return (
      <View style={styles.container}>
        <TextInput
          style={{ height: 40, borderColor: 'gray', borderWidth: 1 }}
          onChangeText={(text) => this.setState({ text })}
          placeholder="Enter search keyword"
        />
        <ListView
          dataSource={this.state.dataSource}
          renderRow={this.renderRow}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    backgroundColor: '#F5FCFF',
    marginTop: 25
  },
  listItem: {
    margin: 10
  }
});
```

在构造函数中实例化ListView.DataSource，传入包含两个元素的数组作为参数。在render方法中声明ListView组件，将状态dataSource赋值给它，并定义renderRow方法，用来渲染视图的每一行。

同时还声明了TextInput组件，为它指定一些基础样式以及占位文字。当输入框的内容发生变化时，onChangeText方法就会被调用。




## 1.10　第二步：与服务器/后端通信

基本的UI已经编写好，现在要调用API来获取真实数据，在ListView组件中进行渲染。之前提到过，网络请求的代码可以放在独立的JavaScript文件中，不过现在只需调用一个API，所以暂时也把这些代码放在主文件中。

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Text,
  View,
  ListView,
  TextInput,
  Image
```

```
} from 'react-native';
import { debounce } from 'lodash';

export default class main extends Component {

  constructor(props) {
    super(props);
    const ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
    this.state = {
      dataSource: ds.cloneWithRows([])
    }
    this.searchMovies = this.searchMovies.bind(this);
  }

  searchMovies = debounce(text => {
    fetch('http://www.omdbapi.com/?s=' + text)
      .then((response) => response.json())
      .then((responseData) => {
        if ('Search' in responseData) {
          console.log(responseData.Search);
          const ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
          this.setState({
            dataSource: ds.cloneWithRows(responseData.Search)
          })
        }
      })
      .catch((err) => {
        console.log(err);
      })
  }, 500);

  renderRow(row) {
    return (
      <View style={styles.listItem}>
        <Image source={{uri: row.Poster}} style={styles.poster} />
        <View style={{flex: 1}}>
          <Text style={styles.title}>{row.Title}</Text>
          <Text style={styles.subHeading}>{row.Type} - {row.Year}</Text>
        </View>
      </View>
    )
  }

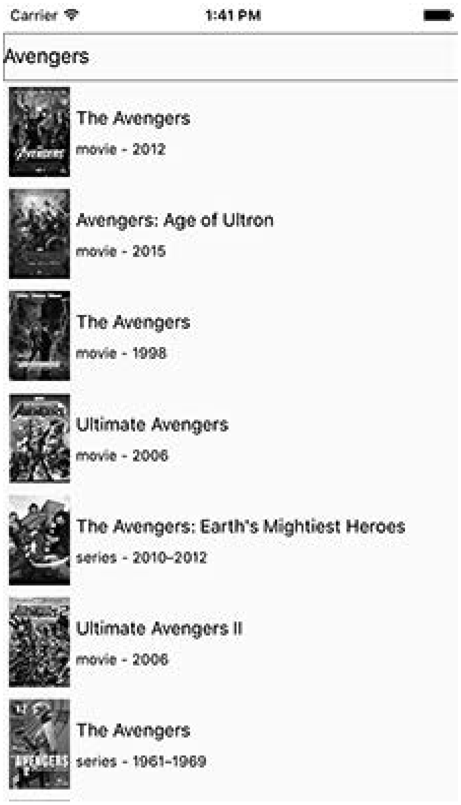
  render() {
    return (
      <View style={styles.container}>
        <TextInput
          style={{height: 40, borderColor: 'gray', borderWidth: 1}}
          onChangeText={this.searchMovies}
          placeholder="Enter search keyword"
        />
```

```
        <ListView
          dataSource={this.state.dataSource}
          renderRow={this.renderRow}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    backgroundColor: '#F5FCFF',
    marginTop: 25
  },
  listItem: {
    flexDirection: 'row',
    alignItems: 'center',
    margin: 5
  },
  poster: {
    height: 75,
    width: 50
  },
  title: {
    margin: 5,
    fontSize: 15
  },
  subHeading: {
    margin: 5,
    fontSize: 12
  }
});
```

应用的逻辑很简单，当用户在TextInput组件中输入文本时，就调用API获取和输入文本匹配的影片。因此，我们修改TextInput组件的代码，把searchMovies方法绑定到onChangeText属性上。

在searchMovies函数中，用debounce方法对函数体进行了一层封装，这样就可以等用户输入完毕后再调用API。需要注意的是，要事先用npm安装第三方库lodash，再从lodash中导入debounce方法。debounce方法中，fecth方法用输入文本作为关键词参数，向http://www.omdapi.com发起请求。响应结果先被解析成JSON，最终得到JSON对象responseData。现在要做的就是把ListView组件已有的datasource属性用新返回的结果数组进行替换。完成这一步后，组件会被重新渲染，我们就能看到符合搜索关键词的影片列表。




## 1.11　第三步：添加动画效果

下面来看如何给ListView组件添加简单的滑入动画。首先将列表每行的内容抽离成独立的组件，命名为ListItem，并且用它替换主组件中的renderRow函数。现在renderRow函数的代码如下所示。

```
renderRow(row, sId, rId) {
  return (<ListItem row={row} delay={rId * 50} />)
}
```

我们为ListItem组件添加了两个属性，row属性包含渲染该行的实际数据，另一个delay属性用来控制每行的动画逐行延迟发生。现在来看一下ListItem组件的代码。

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Text,
  View,
  Image,
```

```
  Animated
} from 'react-native';

export default class listitem extends Component {

  constructor(props) {
    super(props);
    this.state = {
      slideAnim: new Animated.Value(100)
    }
  }

  componentDidMount() {
    Animated.timing(this.state.slideAnim, {
      toValue: 0,
      duration: 500,
      delay: this.props.delay
    }).start();
  }

  render() {
    return (
      <Animated.View style={{marginLeft: this.state.slideAnim, flexDirection: 'row', alignItems:
'center'}}>
        <Image source={{ uri: this.props.row.Poster }} style={styles.poster} />
        <View style={{ flex: 1 }}>
          <Text style={styles.title}>{this.props.row.Title}</Text>
          <Text style={styles.subHeading}>
            {this.props.row.Type} - {this.props.row.Year}
          </Text>
        </View>
      </Animated.View>
    )
  }
}

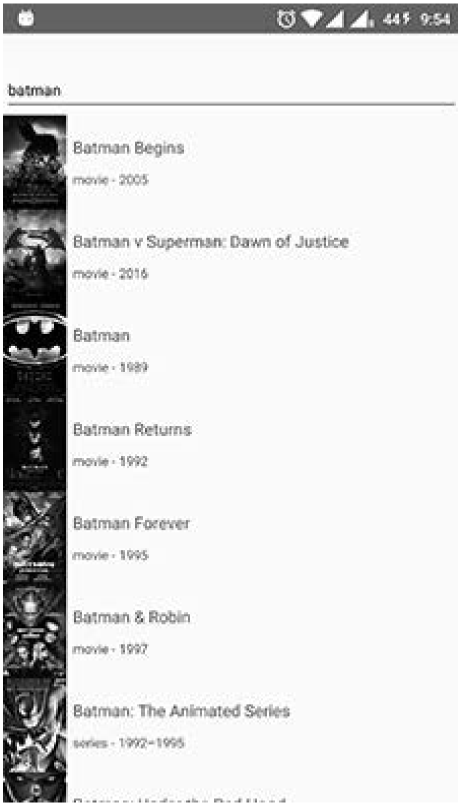
const styles = StyleSheet.create({
  poster: {
    height: 75,
    width: 50
  },
  title: {
    margin: 5,
    fontSize: 15
  },
  subHeading: {
    margin: 5,
    fontSize: 12
  }
});
```

构造函数中用`Animated.value(100)`初始化了一个新状态`slideAnim`。组件挂载后，`Animated.Timing`方法先延迟组件属性中指定的时间，接着在500毫秒内将`slideAnim`的值从100逐

渐减至0。最后，在`render`方法中声明`Animated.View`组件，并把`slideAnim`的值赋给`marginLeft`样式。因此当组件开始渲染时，每个`ListItem`组件的`marginLeft`值会从100变为0，列表的每一行逐渐从右往左移入的过程就构成了连续的滑入动画。

## 1.12　Android 平台上的做法

至此，我们已经成功完成了第一个iOS平台的React Native应用。接下来做点有趣的，启动Android模拟器，把index.ios.js文件的代码复制粘贴到index.android.js中，然后在应用目录下运行`react-native run-android`命令。React Native会在模拟器中安装并启动应用。一切就绪后，就会见到如下图所示的内容。




我们的应用在Android平台上也能运行，代码完全一样，一行都没有修改！网络请求、`ListView`组件、`TextInput`组件还有动画，所有功能都运行得很好。多亏了React Native，我们一下子就同时开发出iOS和Android两个平台的应用，并且都是真正用原生来渲染的。

## 1.13 第四步：添加原生模块

React Native的设计方式使得它可以通过原生模块调用宿主平台的API。想象一下，你想实现Android平台新提供的Material Design风格组件`Snackbar`，可你知道React Native还没有提供对应的原生模块。我们可以自行开发这个模块，并在JavaScript层中调用它。接下来的几章中会详细讨论原生模块的开发。

添加原生模块时会遇到问题：这些模块只针对特定的平台，不能跨平台使用。你刚刚看到我们的应用在iOS和Android平台上都运行得很好，但如果要在应用中使用Android原生的`Snackbar`组件呢？在应用中添加原生组件后，它就不能同时在iOS和Android平台上运行。这种情况下可以先检测应用当前所运行的平台：如果是Android平台，就调用`Snackbar`组件；如果是iOS平台，就弹出一个简单的提示框。

```
var { Platform } = React;

…
showMessage() {
  if (Platform.OS === 'ios') {
    // 显示一个简单的提示框
  } else {
    // 显示自定义的android Snackbar模块
  }
}
…
```

应用在iOS系统的设备或模拟器中运行时，`Platform.OS`属性为字符串`ios`。同理，在Android系统中，该属性为字符串`android`。

同样的思路也可以用在样式方面，`Platform.select`方法接受一个对象参数，以`Platform.OS`属性值作为键名，它就会返回与当前运行平台对应的值。

```
var { Platform } = React

var styles = StyleSheet.create({
  container: {
    flex: 1,
    ...Platform.select({
      ios: {
        backgroundColor: 'red',
      },
      android: {
        backgroundColor: 'blue',
      },
    }),
  },
});
```

上面代码的效果就是让一个容器元素在两个平台上都拥有`flex: 1`样式，而iOS的背景色为红

色，Android的背景色为蓝色。

由于Platform.select方法接受任意的参数值，可以用它返回对应平台的组件，代码如下所示。

```
var Component = Platform.select({
  ios: () => require('ComponentIOS'),
  android: () => require('ComponentAndroid'),
})();
<Component />;
```

这样，我们可以编写在iOS和Android平台上都能运行的组件，也就达成了两个平台共享代码的目的。

### 小结

我们已经用React Native开发了一个简单的应用，用的都是React Native跨平台的核心组件，因此应用可以兼容iOS和Android两个平台。我们不需要任何Objective-C/Swift或者Java语言的知识，只需要JavaScript和React Native就足够了。

## 1.14　部署第一个应用

应用开发完成后，就可以把它发布到Apple应用商店或者Android应用商店，让全世界的用户都可以使用它。

### 1.14.1　部署

React Native应用的部署方式与原生应用的部署方式几乎完全一样。iOS平台要生成JavaScript文件包，并且要用React Native的打包器对静态资源进行打包，以便应用能够加载离线文件包。文件打包完毕后，我们在AppDelegate.m文件中添加几行代码注释，如下所示。

```
// jsCodeLocation =
// [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle"];
```

你可以在模拟器中再运行一次应用，确保它能够正常使用。

现在，可以使用XCode来构建以及发布应用了。请先确保你有Apple的开发者账号，这是在Apple应用商店上发布应用的必备条件。你也可以使用TestFlight服务分发iOS应用来测试beta版本。

发布Android应用时，需要对生成的APK安装包进行签名，然后才能提交给应用商店。与iOS平台一样，在Google应用商店上发布应用需要拥有Google开发者账号。

对APK文件进行签名，首先需要使用keytool生成签名密钥，有了它就可以对应用签名。准备好密钥文件后，把它复制到android/app目录下。然后配置build.gradle使用这些签名密钥来生成APK文件，具体做法如下所示。

```
…
signingConfigs {
    release {
        storeFile file(RELEASE_STORE_FILE)
        storePassword RELEASE_STORE_PASSWORD
        keyAlias RELEASE_KEY_ALIAS
        keyPassword RELEASE_KEY_PASSWORD
    }
}
buildTypes {
    release {
        signingConfig signingConfigs.release
        …
    }
}
…
```

签名过的APK文件生成后，就可以把它提交给Play Store等待审核。

上述做法中，每当你想要发布应用的更新版本时，都要不断重复同样的过程，而且要等待应用市场对更新包进行审核。有了React Native，就可以通过CodePush服务（微软的开源项目）为应用推送动态更新。

### 1.14.2　CodePush

> CodePush插件让应用的JavaScript代码以及图片资源与发布到CodePush服务器上的更新保持同步，用户可以马上看到你对产品所做的改进。你的应用既能受益于离线应用的体验，又能像Web平台一样灵活，让更新包完成后马上就能下载。这是双赢的做法！

CodePush通过与服务端同步的形式，更新应用内打包的JavaScript代码以及静态资源文件。当我们对JavaScript代码或者静态资源做了任意修改后，都要重新生成JavaScript文件并推送到CodePush服务器。只要用户打开应用，CodePush就会检查我们推送的任意更新版本。如果发现了新版本，CodePush会下载JavaScript文件对已有的打包文件和静态资源文件进行更新。这样用户几乎可以马上获取到应用的更新，而且一切都在后台进行，无需用户自己手动操作。

需要留意的关键之处在于，任何涉及原生代码（如修改了AppDelegate.m或MainActivity.java文件）的产品改动无法通过CodePush来分发，因此这种情况要通过相应平台的应用商店进行更新。

## 1.15　总结

本章介绍了什么是React Native，它和现有技术的差异以及它的重要性。还学习了关键的概念、架构以及React和React Native的工作原理。并成功地用React Native开发了第一个真正的跨平台原生iOS和Android应用。

接下来的章节将为你详细介绍一些React Native的概念。

第 2 章

# 原生模块与组件

原生用户界面是React Native的重要特性。运行`react-native init`命令，就会启动一个包含简单界面的Hello World应用，它完全由原生视图以及文本组件来渲染。JavaScript编写的React组件仅仅作为原生视图的抽象表现以及配置。所有React组件最后都会被渲染成原生UI组件。我们可以在其他组件的基础上开发React组件，同时这些组件本身也是用另外的组件开发的。然而从组件关系链的末端来看，我们只是编写规则让原生层接收并显示出来。

尽管React Native通过我们所熟悉的JavaScript和React让用户界面开发变得很简单，移动应用还是需要经常调用原生API来获取数据，比如设备旋转方向、当前地理位置、网络状态、电量水平等。React Native提供的库可以解决上述需求，它包含了许多暴露为`NativeModules`的API，但其中可能缺少你所需要的API，或者你想在原生端定制的业务逻辑并不适合让React Native的核心库来提供。开发者可以用React Native创建自己的原生API，让应用的JavaScript层可以调用。这些原生模块可以直接用在Android或iOS的SDK中，它们的概念类似于NodeJS的原生`c++`模块，通过`node-gyp`绑定进行编译，我们通过React Native库提供的抽象能够与Java或Objective-C/Swift进行交互，而不是与V8这样的底层JavaScript引擎打交道。库所暴露的可扩展原生模块类与桥接层交互，桥接层负责应用两个层级间的序列化、反序列化以及批处理通信。通过Android的函数注释以及iOS的宏定义，你编写的方法可以让JavaScript调用，或者传播事件让JavaScript监听。原生模块函数从JavaScript端接收参数，经过桥接层后，参数被处理成恰当的类型并且可以直接使用。

本章将介绍原生组件和原生模块的架构方式，以及如何开发它们。

## 2.1　第一个原生组件

你首先要接触的原生组件很可能是`View`和`Text`组件，它们直接映射到对应的原生组件上，即iOS系统的`UIView`和`UILabel`以及Android系统的`android.view.View`和`android.widget.TextView`。当定义一个组件来渲染一个包含几行文本的`View`时，原生层会建立与该界面结构一样的镜像。桥接层把规则（组件属性）从UI层传递给原生层。最终在屏幕上看到的，就是React组件转译成相应原生组件的结果。我们将深入探究“视图管理器”如何管理原生UI组件，以及React组件属性如何传递给在原生层定义的函数并选择处理方式。

接下来要介绍的主要概念是，每一个JSX元素都与其所代表的原生组件的实例绑定到一起。两者总是与对方保持同步，React组件定义页面结构，原生UI组件就渲染出对应的UI结构。

```
class HelloWorld extends React.Component {
  render() {
    return (
      <View>
        <Text>Hello World!</Text>
      </View>
    )
  }
}
```



JavaScript层的代码很直观，上述代码声明了包含几行文本的视图。然而为了实现两层之间的同步，实际上有很多工作要做。HelloWorld组件挂载之后，通知原生层展现一个View节点并包含一个Text节点。UIView（或android.view.View）以及UILabel（或android.widget.TextView）的实例被创建。每个实例分配了一个标识符，并与JavaScript层共享。从此刻起，只要这些视图的属性发生了改变，或者将它们从屏幕上移除，这些改变集合会根据它们的身份（组件ID）和改变内容（任意属性）通知给原生层，原生层就会按照JavaScript层传来的信息调整布局。

最后，React Native中的JSX语法类似于我们所熟悉的React虚拟DOM，它就是用来编写设置原生视图的配置文件。

## 2.2　剖析原生组件

一个原生组件主要由两部分组成：ViewManager，以及实际的UI组件。UI组件的类，可以继承自某个Android的视图，或者利用iOS的RCTView作为特殊容器来封装任意的视图控制器。

ViewManager扮演的角色负责连接React组件以及该组件代表的原生UI组件实例。ViewManager是单例模式，对于指定的组件类型只有唯一一个视图管理器，它管理着指定类型的所有组件。

React组件挂载之后就会触发它的render方法，并返回它所维护的JSX组件树。这些树节点根据它们的类型被发送给对应的ViewManager（比如<View />节点会被发送给处理View实例的管理器）。元素的视图管理器创建出React组件代表的原生组件实例。如果该组件拥有属性，这些属性将由ViewManager的函数接收，用来创建符合这些属性的原生组件实例。接着就可以更新React组件对应的原生组件。当属性发生变化后会进行同样的过程。这里就由开发者来决定接下来要做什么了。假设我们要创建一个自定义地图UI组件，从JavaScript层接收了一些经纬度坐标数据后，把地图定位到新的中心点。我们很快就会动手开发这个示例，首先来看看对Android原生组件的分析。

假设我们有一些自定义原生UI类称作CustomView。由于React Native中已经有完全可用的View组件，并且可配置，生产实践中我们永远不需要写下面的代码，现在出于讨论的需要暂时这样写。

```
class CustomView extends View {
  ...
}
```

出于某种需要，我们要让CustomView显示在屏幕上。一些React组件将会控制它的显示方式，比如显示成<CustomView />元素。

每个原生组件都需要一个视图管理器。CustomViewManager继承SimpleViewManager并指定属于它的原生UI组件CustomView。

```
/**
 * 控制CustomView的CustomViewManager类
 */
class CustomViewManager extends SimpleViewManager<CustomView> {
  @Override
  public String getName();

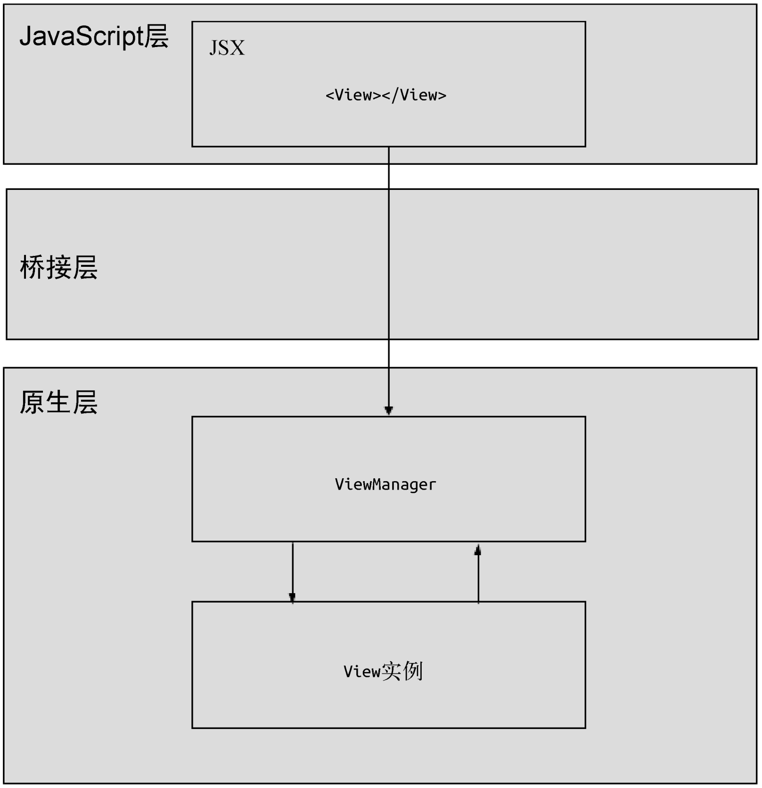
  @Override
  protected CustomView createViewInstance (ThemedReactContext context);

  @ReactProp(name = "someProp")
  public void setSomeProp(CustomView customViewInstance, @Nullable String value);
}
```

我们实现了两个继承自父类的方法getName和createViewInstance，以及自定义方法setSomeProp。

- getName方法返回组件名称，在JavaScript层被引用。
- createViewInstance方法用于在JavaScript层挂载React组件时创建CustomView的实例。
- setSomeProp方法在React组件属性包含初始值或新值时被调用。参数为相应的实例和属性值。

总体上看控制流从JavaScript层进入原生层。React组件负责渲染JSX组件，JSX组件的属性以及层次结构通过桥接层传递给原生层。相应的ViewManager取得这些数据后，如果不存在已有实例，就调用createViewInstance方法创建一个。每个原生元素都有一个标识符，创建React组件的时候需要提供对应的标识符。这便是React Native能够映射两个层次结构的奥秘所在。如果JSX组件定义了某些属性，它将寻找对应的函数（带有@ReactProp和@ReactPropGroup注释），并以相应的视图实例和属性值作为参数调用该方法。根据JavaScript组件定义的配置和布局，屏幕上就会渲染出最终的原生组件。




下面来逐步分析。假设应用只包含一个组件，并满足以下条件。

(1) 组件是一个简单的View组件。

(2) 初始背景色为红色。

(3) 几秒后背景色变为黄色。

```
class BackgroundExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      backgroundColor: 'red'
    };
  }
  componentWillMount() {
    setTimeout(() => this.setState({backgroundColor: 'yellow'}), 3000);
  }
  render() {
    var {backgroundColor} = this.state;
    return (
      <View style={[styles.root, {backgroundColor}]}></View>
    );
```

```
  }
}

const style = StyleSheet.create({
  root: {
    flex: 1
  }
});
```

应用启动并挂载组件后，渲染JSX组件传递给原生层。在原生层找到`<View />`组件的视图管理器，并通过它调用`createViewInstance`方法，初始化`android.view.View`（iOS系统为`UIView`）并返回。React Native内部为该组件创建一个`ShadowNode`，这个`ShadowNode`与JavaScript层的React组件配对，两者从此保持同步。

`<View />`元素也有一个`style`属性，包含了`flex`和`backgroundColor`样式规则。这些值会带上视图的ID，通过桥接层传递来通知React Native找到需要修改的原生视图实例。接着该视图实例和样式属性值一起传给负责处理样式规则的方法。在这个方法内，原生视图会被处理成弹性布局，同时更新它的背景色。

3秒后状态发生了改变，`backgroundColor`的值变成了`'yellow'`。状态的改变导致组件的`render`方法接着被调用，返回的`View`元素更新了`style`属性，变成了`{backgroundColor: 'yellow'}`。新值发送给`ViewManager`，我们知道，此时它将根据新值对原生视图的背景色进行更新。用户看见的效果就是背景色从红色变成了黄色。

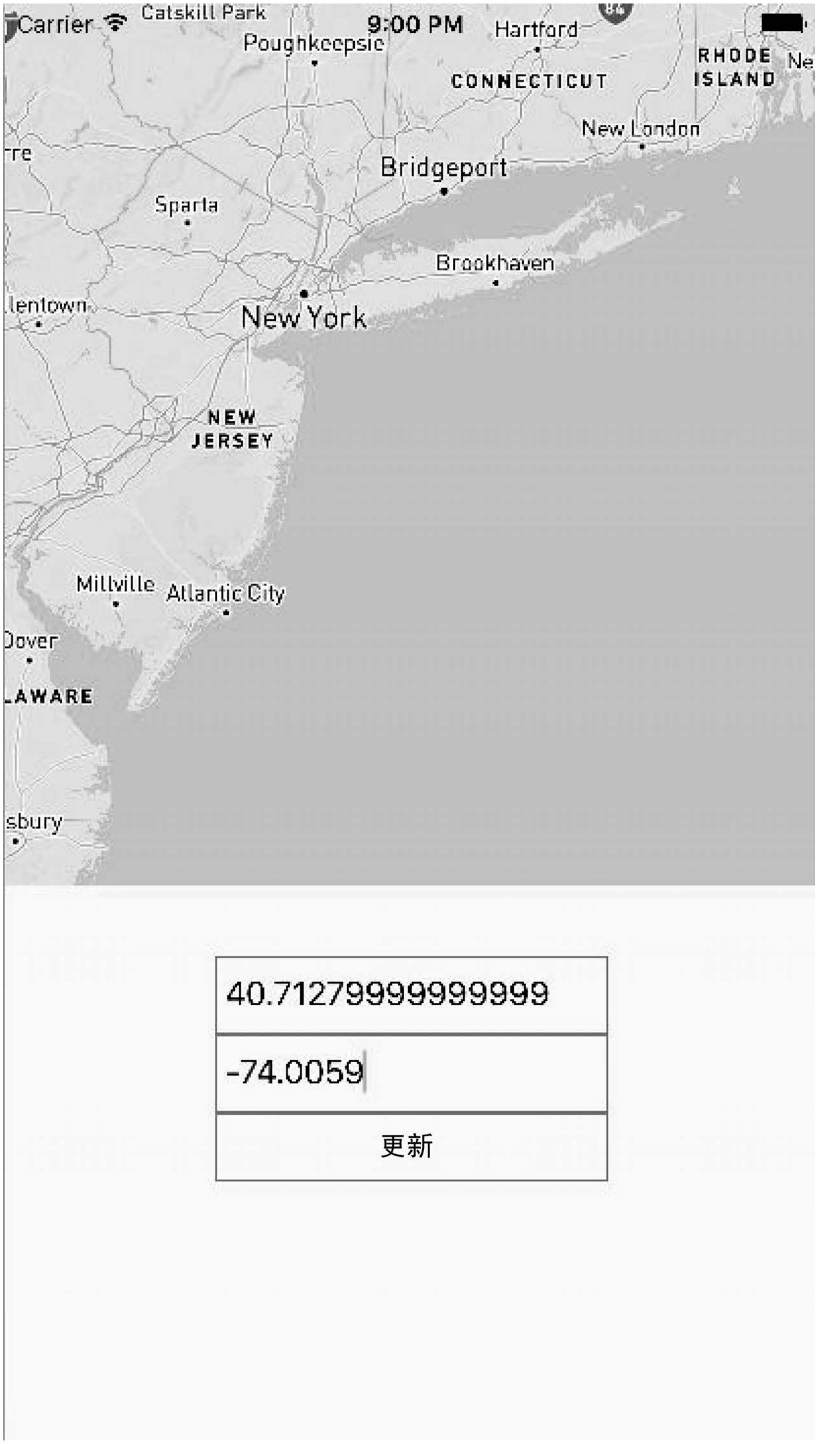
以上便是React Native组件的主要概念。每个JavaScript React节点都有配对的原生节点。对每个应用节点执行对应的操作：绑定唯一标识符。React节点所发生的一切都会根据标识符和修改后的属性，转换成原生节点。这两者始终互相保持同步。既然我们已经理解了原生组件各部分的原理，接着就来动手创建一个。

## 2.3　创建自定义原生组件

在接下来的示例中将用MapBox的SDK开发地图组件。最终完成的React Native组件包含如下结构。

```
<MapBoxView
  zoom={10}
  center={{latitude, longitude}}
  onCenterChanged={this.onCenterChanged}
/>
```

我们开发的界面上将包含两个输入框、一个按钮和一张地图。输入框负责显示地图的当前坐标，并且用户也可以对它们进行编辑。按钮用来提交坐标并重新定位地图。地图会根据提供的坐标进行定位，当用户移动地图时返回新的坐标并显示在输入框中。


Carrier
9:00 PM
Catskill Park
Poughkeepsie
Hartford
CONNECTICUT
RHODE ISLAND
New London
Bridgeport
Sparta
Brookhaven
New York
NEW JERSEY
Millville
Atlantic City
40.712799999999
-74.0059
更新

先来编写React组件。

```
class MapDemo extends React.Component {
  constructor(props) {
    // 记得传入props
    // 我们终究希望原生层知道最初的状态
    super(props);

    // 起始坐标经纬度均为0
    var center = {latitude: 0, longitude: 0};

    this.state = {
      // 地图中心
      mapCenter: {
        ...center
      },
      // 编辑后的坐标（避免在用户输入时更新地图）
      editCenter: {
        ...center
      }
    };
  }

  /**
   * 地图位置移动后更新输入框
   */
  onCenterChanged({latitude, longitude}) {
    var editCenter = {latitude, longitude};
    this.setState({editCenter});
  }

  /**
   * 提交输入框内容后更新地图
   */
  updateMapCoordinates() {
    var latitude = parseFloat(this.state.editCenter.latitude);
    var longitude = parseFloat(this.state.editCenter.longitude);
    var mapCenter = {latitude, longitude};
    this.setState({mapCenter});
  }

  /**
   * 用户输入时更新单个输入框的内容
   */
  updateEditField(plain, value) {
    var editCenter = {
    ...this.state.editCenter,
    [plain]: value
  };
  this.setState({editCenter});
  }

  render() {
    return (
```

```
      <View>
        <MapBoxView
          zoom={10}
          center={this.state.mapCenter}
          onCenterChanged={event => this.onCenterChanged(event.nativeEvent.src)}
        />
        <TextInput
          value={this.state.editCenter.latitude}
          onChangeText={value => updateEditField('latitude', value)}
        />
        <TextInput
          value={this.state.editCenter.longitude}
          onChangeText={value => updateEditField('longitude', value)}
        />
        <TouchableHighlight onPress={() => updateMapCoordinates()}>
          <Text>Update</Text>
        </TouchableHighlight>
      </View>
    );
  }
}
```



接下来分别看看Android和iOS的情况，每个平台都会为MapBox的MapView元素实现一个视图管理器。

## 2.3.1 Android

视图管理器类一开始从SimpleViewManager继承最初的结构并实现所需的方法。上文提到过，它是单例模式的实例，任何MapBoxView组件的通信都要经过它。屏幕上新增了MapBoxView组件，就会创建该视图管理器。组件的任何属性发生了改变，比如坐标，也会由它来处理。

```
class ReactMapBoxViewManager extends SimpleViewManager<MapView> {
  public static final String REACT_CLASS = "ReactMapBoxView";
  private String accessToken;

  ReactMapBoxViewManager(String accessToken) {
    this.accessToken = accessToken;
  }

  /**
   * React组件的名称
   */
  @Override
  public String getName() {
    return REACT_CLASS;
  }

  /**
   * 创建新的MapView实例
   * 该过程在新的React组件MapBox挂载到应用的时候执行
   */
```

```
  protected MapView createViewInstance(ThemedReactContexted themedReactContext) {
    MapView mapView = new MapView(themedReactContext, accessToken);
    // 必需的调用
    mapView.onCreate(null);
    return mapView;
  }
}
```

SimpleViewManager<MapView>负责通知父类所要管理的View类型，MapView.createViewInstance方法初始化MapView并返回React组件代表的实例。这里需要注意一点，MapView构造函数需要一个访问令牌作为参数。不巧的是，createViewInstance方法处于我们控制范围以外的流程中，而且此时还不能访问组件的属性，因此把访问令牌设置为React组件的属性并没有什么用。这种情况下可以考虑一个巧妙的解决方案，不过这个示例中采用的方式很直接。为了能在createViewInstance方法中使用令牌，在我们拥有控制权的ReactMapBoxViewManager构造函数中，把令牌作为参数的一部分。后面将详细解释在ReactPackage中对此进行初始化的过程发生在何处，以及在何处可以获得直接操作的权限，并提供一个令牌给MapView使用。

至此我们已经拥有了一个能够工作的组件，但是还不能提供任何特殊的配置。接下来添加一些函数对center和zoom属性进行处理。

```
class ReactMapBoxViewManager extends SimpleViewManager<MapView>{
  ...
  public static final String REACT_PROP_CENTER = "center";
  public static final String REACT_PROP_ZOOM = "zoom";
  ...

  @ReactProp(name = REACT_PROP_CENTER)
  public void setCenter(MapView view, @Nullable ReadableMap center) {
    if (center != null) {
      double latitude = center.getDouble("latitude");
      double longitude = center.getDouble("longitude");
      CameraPosition cameraPosition = new CameraPosition.Builder()
        .target(new LatLng(latitude, longitude))
        .build();
      view.moveCamera(CameraUpdateFactory.newCameraPosition(cameraPosition));
    }
  }

  @ReactProp(name = REACT_PROP_ZOOM)
  public void setZoom(MapView view, double zoomLevel) {
    view.setZoom(zoomLevel);
  }
}
```

现在我们的组件已经就绪，可以接收一些配置参数了。用@ReactProp注释的方法负责组件属性交互。该注释的第一参数name是React组件的属性名称。它所注释的方法接受两个参数：MapView的实例和属性值。只需定义好哪个函数负责处理哪个属性，React Native内部就会完成其余的工作。

回到React这一层，我们有如下代码。

```
<MapBoxView center={{latitude: 0, longitude: 0}} />
```

这个React组件会在某种方式下被转译成原生层的代码。MapBoxView将映射到ReactMapBoxViewManager上，它有一个以@ReactProp(name = "center")进行注释的方法。根据这种映射关系，React内部就可以作出以下判断。

(1) 调用哪个方法。

(2) 该方法需要的参数类型是什么。

(3) React组件所配对的原生MapView实例会调用setCenter，属性参数的类型为ReadableMap，它本质上是类型安全的Map数据结构（例如：通过center.getDouble的形式读取数据）。接着从ReadableMap中取出数据建立新位置坐标。通过MapBox SDK提供的CameraPosition类来创建新的位置目标，并更新MapView实例。用户就看到地图移到了新的位置。



在第一步中，注册回调函数处理用户移动地图的操作。

```
private static final String EVENT_REGION_CHANGED = "onRegionChanged";
...

@Override
@Nullable
public Map getExportedCustomDirectEventTypeConstants() {
  MapBuilder.Builder builder = MapBuilder.builder();
  builder.put(EVENT_REGION_CHANGED, MapBuilder.of("registrationName",
EVENT_REGION_CHANGED));
  return builder.build();
}

@ReactMethod(name = EVENT_REGION_CHANGED, defaultBoolean = true)
public void onRegionChangedListener(final MapView map, Boolean value) {
  view.addOnMapChangedListener(new MapView.onMapChangedListener() {
    @Override
    public void onMapChanged(int change) {
      if (change == MapView.REGION_DID_CHANGE ||
          change == MapView.REGION_DID_CHANGE_ANIMATED) {
        WritableMap event = Arguments.createMap();
        WritableMap location = Arguments.createMap();

        location.putDouble("latitude", view.getCenterCoordinate().getLatitude());
        location.putDouble("longitude", view.getCenterCoordinate().getLongitude());
        location.putDouble("zoom", view.getZoom());
        event.putMap("src", location);

        ReactContext reactContext = (ReactContext) view.getContext();
        context.getJSModule(RCTEventEmitter.class)
          .receiveEvent(view.getId(), EVENT_REGION_CHANGED, event);
      }
    }
  });
}
...
```

分发到React组件并绑定的回调函数需要一些额外的配置。毕竟，React Native需要一种策略把任意的事件传播给绑定在React组件属性上的函数，这可绝非易事。此处就要用到getExportedCustomDirectEventTypeConstants方法，它负责定义事件、组件，以及绑定了回调函数的组件属性之间的映射关系。它接受事件名称以及一个映射值registrationName:callbackPropertyName作为参数。这样React Native在分发事件给组件时就知道要查找的属性名称。

下面这段话来自源代码文档。

> 配置数据的映射关系被返回给JavaScript层，由此定义了可以用在原生视图上的合法事件，这种类型的事件应当直接分发，并且不需要冒泡。

以下代码会创建事件映射。

```
"onRegionChanged": {
  "registrationName": "onRegionChanged"
}
```

上述配置的实际意义如下所示。

```
"nameOfEvent": {
  "registrationName": "nameOfPropertyOnComponentWithCallback"
}
```

在这层映射关系下，组件属性onRegionChanged有对应的ReactProp注释。需要注意的是，没有必要为事件回调属性创建原生属性处理函数。我们可以把监听器以及事件触发器绑定到映射视图上，作为createViewInstance方法的一部分，因此不需要在原生层创建一个属性来实现这样的目的。然而如果映射事件没有回调函数，我们也没有必要把它传播给组件。通过把监听器设置成属性处理函数的一部分，就能够避免在没有设置属性的情况下绑定监听器，因为只有当React组件拥有onRegionChange属性时，属性处理函数才会被调用。

该函数内部为MapView实例绑定了MapView.onMapChangeListener监听器。当监听到移动地图的事件时，就创建一个包含新坐标的对象。event变量用于存放事件对象，location变量用于存放要与事件一起发送的数据，两者都是WritableMap类型。

**注意：JavaScript层接收了来自原生层的事件后，原生事件对象位于JavaScript事件对象的nativeEvent属性中。**

创建了event和location对象之后，可以通过RCTEventEmitter类把操作分发到JavaScript层。回到createViewInstance方法中，我们用当前的ThemedReactContext（ReactContext的子类，ReactContext是Context的子类）实例化MapView。为了访问其他模块，比如RCTEventEmitter，需要从view对象中获取当前的执行环境。通过暴露出来的ReactContext，就可以获取其他React Native模块。这样一来，我们取得了RCTEventEmitter并接着调用receiveEvent。这个过程会利用取得的UI组件的唯一标识符，把UI组件与对应的React组件进行映射。记住，每个原生节点都会根据自身的唯一标识符，与JavaScript层的React节点配对，这就是React判断事件分发目标的方式。

第二参数传入事件的名称，最后一个参数传入事件对象本身。由于我们用getExportedCustomDirectEventTypeConstants方法定义了事件与组件属性之间的映射关系，React Native就能把该事件分发给对应的React组件，组件属性onRegionChanged上注册的事件回调就会被触发。

现在已经配置好了原生组件，可以接收属性以及分发事件了。JavaScript层还需要几个步骤为原生组件定义React接口。不过接下来先讨论如何集成iOS系统的支持。如果你对iOS的章节不感兴趣，可以直接跳到之后JavaScript的部分。

### 2.3.2　iOS



iOS系统对原生模块的实现有几点不同。首先，地图视图要放在“React视图宿主”之中。这是专为React Native开发的视图容器，它可以托管任何自定义的原生视图。如果想要使用自定义的原生视图，只要把它添加成RCTView宿主的子视图。除此以外就是视图管理器的使用了，与Android系统上的做法一样。创建RCTViewManager的子类，它与Android系统的SimpleViewManager扮演一样的角色，JavaScript层React组件所表示的视图由它来控制和操作。

```
// ReactMapBoxViewManager.h
#import "RCTViewManager.h"

@interface ReactMapBoxViewManager : RCTViewManager
@end

// ReactMapBoxViewManager.m
#import "ReactMapBoxViewManager.h"

@implementation ReactMapBoxViewManager

RCT_EXPORT_MODULE();

-(UIView *)view
{
  // 很快就会补上此处的代码
}

@end
```

头文件中实现了RCTViewManager类，除此以外该文件中没有其他内容。文件内部的实现包含RCT_EXPORT_MODULE宏定义，让React Native的其他部分可以访问该类。view方法将用于创建map视图的实例（如同Android系统的createViewInstance），我们先搭建完自定义的RCTView容器，再回过头来介绍它。

接下来的步骤要创建一个继承自RCTView的自定义视图组件。它将用来托管MapBox地图并与视图管理器交互。

```
// ReactMapBoxView.h
#import <Mapbox/Mapbox.h>
#import "RCTView.h"
```

```
@interface ReactMapBoxView : RCTView

-(void)setCenter:(CLLocationCoordinate2D)center;
-(void)setZoom:(double)zoomLevel;

@end
```

在头文件中，ReactMapBoxView继承RCTView类并声明了两个方法：setCenter和zoomCenter。它们将负责处理center和zoom属性值。

```
// ReactMapBoxView.m
#import "ReactMapBoxView.h"

@implementation ReactMapBoxView {
  // 地图实例
  MGLMapView * _map;
}

// MapView有个问题，它不能创建没有边界的地图
// 要等待视图初始化完毕之后再加载地图
-(void)createMap
{

  // 初始化新的地图
  _map = [[MGLMapView alloc] initWithFrame:self.bounds];

  // 用RCTView容器调整地图尺寸
  _map.autoresizingMask = UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight

  // 把地图添加为RCTView的子视图
  [self addSubview:_map];
}

-(void)setCenter:(CLLocationCoordinate2D)center
{
  if (!_map) {
    [self createMap];
  }
  // 设置地图中心位置
  _map.centerCoordinate = center;
}

-(void)setZoom:(double)zoomLevel
{
  if (!_map) {
    [self createMap];
  }
  // 设置地图缩放等级
  _map.zoomLevel = zoomLevel;
}
```

继承RCTView的ReactMapBoxView才是真正的地图视图的容器。我们在一个方法中创建地图，

设置容器边界为地图的框架。该方法还确保地图尺寸可以随着宿主视图的尺寸自由调整。最后，把地图添加为宿主视图的子视图。

另外两个方法根据React组件属性对地图进行控制。这里与Android系统有一些不同，Android的视图管理器负责大部分属性的处理逻辑。iOS系统的视图管理器好比属性的代理或寄存器，实际的视图会通过宏定义施展的小魔法来接收并处理这些属性。

为了最终把两个部分结合起来，下面回到ReactMapBoxViewManager的代码，初始化这个自定义的视图，并建立属性映射关系。

```
// ReactMapBoxViewManager.m

 #import "RCTMapBoxViewManager.h"
+#import "ReactMapBoxView.h"

 @implementation RCTMapBoxViewManager

 RCT_EXPORT_MODULE();

 -(UIView *)view
 {
+ return [[ReactMapBoxView alloc] init];
 }

+RCT_EXPORT_VIEW_PROPERTY(center, CLLocationCoordinate2D);
+RCT_EXPORT_VIEW_PROPERTY(zoom, double);

@end
```

现在，我们完善了view方法，它与Android系统的createViewInstance方法一样，负责初始化并返回ReactMapBoxView。

宏定义RCT_EXPORT_VIEW_PROPERTY用于定义属性映射关系。它接受两个参数：属性名称，期望的值类型。宏定义的魔法就发生在此处，它会根据属性名称，调用view方法返回的ReactMapBoxView的setter方法。以属性center为例，宏定义会调用地图视图实例上的setCenter方法。类似地，zoom属性会根据React元素的属性值调用实例的setZoom方法。

最后，设置React组件的onRegionChanged回调函数。当地图发生移动时，使用桥接层的eventDispatcher与原生组件交互。首先更新ReactMapBoxView，以包含MGLMapViewDelegate协议的实现，该协议用于处理地图事件。另外在构造函数中注入ReactMapBoxViewManager提供的eventDispatcher引用。由于ReactMapBoxView拥有地图视图的实例，它负责事件的处理以及传播。

下面在ReactMapBoxView的头文件中增加一行声明，提供eventDispatcher的注入方式。

```
// ReactMapBoxView.h

 #import <Mapbox/Mapbox.h>
 #import "RCTView.h"
+#import "RCTEventDispatcher.h"
```

```
-@interface ReactMapBoxView : RCTView
+@interface ReactMapBoxView : RCTView<MGLMapViewDelegate>

+-(instancetype)initWithEventDispatcher:(RCTEventDispatcher *)eventDispatcher;
 -(void)setCenter:(CLLocationCoordinate2D)center;
 -(void)setZoom:(double)zoomLevel;

  @end
```

接下来在实现ReactMapBoxView的文件中添加一个eventDispatcher的实例，定义初始化方法，将ReactMapBoxView注册为地图的代理以便接收地图移动事件并分发到JavaScript层。

```
// ReactMapBoxView.m
 #import "ReactMapBoxView.h"
+#import "UIView+React.h"

@implementation ReactMapBoxView {
  // 地图实例
  MGLMapView * _map;

+ // eventDispatcher实例
+ RCTEventDispatcher * _eventDispatcher;
 }

+-(instancetype)initWithEventDispatcher:(RCTEventDispatcher *)eventDispatcher
+{
+  if (self = [super init]) {
+    _eventDispatcher = eventDispatcher;
+  }
+  return self;
+}
+-(void)mapView:(ReactMapBoxView *)mapView regionDidChangeAnimated:(BOOL)animated
+{
+  CLLocationCoordinate2D region = _map.centerCoordinate;
+
+  NSDictionary *event = @{
+                          @"target": self.reactTag,
+                          @"src": @{
+                              @"latitude": @(region.latitude),
+                              @"longitude": @(region.longitude),
+                              @"zoom": [NSNumber numberWithDouble:
_map.zoomLevel]
+                              }
+                          };
+ [_eventDispatcher sendInputEventWithName:@"onRegionChanged" body:event];
+}
```

在regionDidChangeAnimated代码块中，我们取得了当前的地图坐标并构造了一个事件字典。它将会被分发给React组件的回调属性。可以通过事件对象的target属性来识别接收事件的目标组件。target属性通过self.reactTag获取React节点的ID。再强调一次，React组件与原生组件以这种处理方式通过桥接层互相识别。事件对象的src属性包含了表示地图坐标的真实数据。

ReactMapBoxView已经接近完成，很快就能与React组件传递地图变化数据了。为了把所有的工作整合起来，接下来将搭建ReactMapBoxViewManager用于注入事件分发器，并定义应该直接传播给组件本身的事件，以及组件属性上绑定的回调函数。如果你跳过了前面Android系统的章节，应当回头阅读一下原生事件如何调用React组件的回调函数那一部分。

```
// ReactMapBoxViewManager.m

 #import "RCTMapBoxViewManager.h"
 #import "ReactMapBoxView.h"
 #import "RCTBridge.h"
+#import "RCTEventDispatcher.h"

 @implementation RCTMapBoxViewManager

 RCT_EXPORT_MODULE();
 @synthesize bridge = _bridge;

 -(UIView *)view
 {
-  return [[ReactMapBoxView alloc] init];
+  return [[ReactMapBoxView alloc] initWithEventDispatcher:self.bridge.eventDispatcher];
 }

+-(NSArray *)customDirectEventTypes
+{
+ return @[@"onRegionChanged"];
+}

 RCT_EXPORT_VIEW_PROPERTY(center, CLLocationCoordinate2D);
 RCT_EXPORT_VIEW_PROPERTY(zoom, double);

 @end
```

现在，React组件的onRegionChanged回调函数可以接收用户移动地图的事件了。

### 2.3.3　JavaScript

原生组件已经就绪，下面要在JavaScript层为它定义React部分的接口。此处将会定义允许的属性类型。基本来讲，React组件只需要定义组件接口的部分，不需要定义完整的类。

```
// mapBox.js

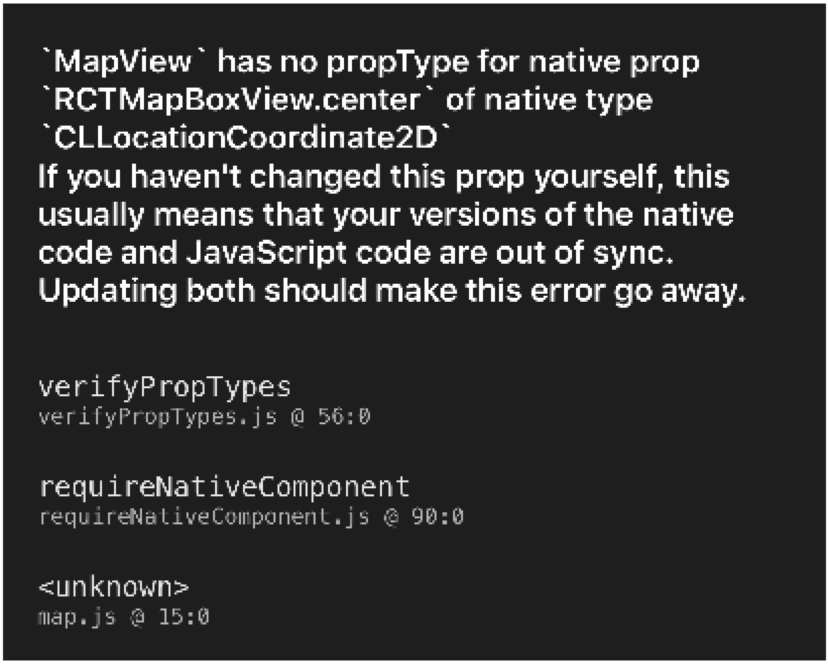
import {PropTypes} from 'react';
import {requireNativeComponent, View} from 'react-native';

// 定义暴露给原生组件的接口
var iface = {
  name: 'MapBoxView',
  propTypes: {
    center: PropTypes.shape({
      latitude: PropTypes.number,
```

```
      longitude: PropTypes.number
    }),
    zoom: PropTypes.number,
    onRegionChanged: PropTypes.func,
    ...View.propTypes
  }
};

// export语句导出的接口，会被原生组件导入成JSX组件来使用
export default requireNativeComponent('ReactMapBoxView', iface);
```

iface对象中定义了组件的接口，包括React组件的名称以及属性类型。任何原生层所定义的属性都必须在接口中定义好它们的类型。如果有任何遗漏，下图所示的红色警告界面会通知你。




requireNativeComponent方法将根据提供的名称，在ViewManager中进行查找。Android系统上要匹配视图管理器的getName()方法，iOS系统要匹配Manager以外的类名或者传给RCT_EXPORT_MODULE宏的可选字符串。

由requireNativeComponent方法准备好并返回的组件，通过这个文件导出。如果其他React组件想要使用该组件，只需导入该文件即可。

```
// screen.js

import React from 'react';
import MapBoxView from './mapBox';
```

```
class Screen extends React.Component {
  render() {
    return (
      <MapBoxView
        center={{
          latitude: 40.7128,
          longitude: -74.0059
        }}
        zoom={8}
        onRegionChanged={event => console.log('region changed', event.nativeEvent.src)}
      />
    );
  }
}
```



现在你已经了解原生组件的构建方式，也学习了两个层级之间如何互相协助来维护一个简单的UI组件展现。总体来看，React Native组件就是原生组件，只不过它还提供了一个代理来实现属性的映射，以及负责与JavaScript层交互的媒介。

## 2.4 原生模块

原生模块就是能够在JavaScript层调用的API。通常来说，开发它们的主要目的在于对两个层级之间的请求进行代理。第1章已经提过，两层之间所有的序列化与批处理请求都由桥接层传输。这意味着对原生模块的全部请求都要异步执行。如果原生方法需要为JavaScript层的调用返回数据，该操作将通过promise或者回调函数来完成。React Native为这两种方式都提供了接口。

对原生模块进行分析之前，先要知道这些模块可以访问原生层的一切。它们就是JavaScript调用Android或iOS SDK的入口。原生模块可以访问activity、运行环境、GPS、存储空间等。实际上你创建的是一个API，JavaScript可以通过它访问设备原生层中任何你所需要的东西。

### 2.4.1 剖析原生模块

iOS和Android对原生模块的具体实现差别非常大，不过总体的思路是相似的，如下所示。

(1) 每个模块继承自原生模块父类。

(2) 每个模块都定义了名称，以便JavaScript层访问。

(3) 每个模块都导出可调用的方法，并包含Java的注释或Objective-C的宏。

#### 1. Android

Android平台的原生模块继承`ReactContextBaseJavaModule`，并且必须实现`getName`方法，它的返回值作为JavaScript模块的名称。

任何允许JavaScript层调用的方法，都带有`@ReactMethod`注释。React的内部机制会把JavaScript

层的请求映射到这些方法上。每个方法的身份由它的签名、名称以及参数进行鉴别。方法所需的任何参数都会被转换成对应的属性类型。

```
public class ExampleModule extends ReactContextBaseJavaModule {
  public ExampleModule(ReactApplicationContext reactContext) {
    super(reactContext);
  }

  @Override
  public String getName() {
    return "ExampleModule";
  }

  @ReactMethod
  public void callableFunction() {
    Log.d(getName(), "Called native function");
  }
}
```

2. iOS

iOS平台的原生模块要么实现了`RCTBridgeModule`协议，要么就是普通的Objective-C类。它们通过`RCT_EXPORT_MODULE()`宏把自身注册成原生模块。这个宏接受一个可选的参数，用来定义JavaScript层的模块名称。默认情况下它会使用类名作为模块名称。所有JavaScript层要访问的方法都由`RCT_EXPORT_METHOD`宏来编写。

```
// ExampleModule.h
#import "RCTBridgeModule.h"

@interface ExampleModule : NSObject <RCTBridgeModule>
@end

// ExampleModule.m
@implementation ExampleModule

RCT_EXPORT_MODULE('ExampleModule');

RCT_EXPORT_METHOD(callableFunction)
{
  RCTLogInfo(@"Called native function")
}
```

如果你正在开发要在两个平台运行的应用，最好尽可能保持API名称一致。如果不可能完全保证一致，那么可以通过平台相关的文件在JavaScript层暴露通用的接口（如yourModule.android.js和yourModule.ios.js）

3. JavaScript

JavaScript层把原生模块作为`NativeModules`对象的一个属性，并且任何带有`@ReactMethod`注释或者属于`RCT_EXPORT_METHOD`宏的方法都能够被调用。

```
import {NativeModules} from 'react-native';
const ExampleModule = {NativeModules};

ExampleModule.callableFunction();
```

### 2.4.2　参数

原生模块的方法和普通方法一样可以接受参数。React Native在提供相关信息方面做得非常好。除了对象或者数组以外的所有参数都有类型注释。在JavaScript代码中传了错误类型的参数或者漏传了某个参数的情况下，都会抛出错误（这点一定要注意）。如果参数是对象或者数组，那么无法保证它们内部的值类型是否正确。



值得注意的是方法签名（包括方法名和参数）。在Java和Objective-C中调用方法一定要带上所定义的参数。JavaScript中不需要这么做。但是当你从JavaScript层调用一个原生方法时，一定要提供方法所期望的全部参数。如果你要开发供其他开发者使用的第三方模块，较好的做法是定义一个JavaScript接口来提供默认值，以防使用者没有传入期望的全部参数。（或者直接抛出异常，因为参数就是应该传入。）另外带有`@ReactMethod`注释的方法不能够重载，每个方法必须拥有独立的名称。

```
@ReactMethod
public void callableFunction (String foo, Integer bar, ReadableMap jsonObject){
  // ...
}

RCT_EXPORT_METHOD(callableFunction:(NSString *)foo bar:(NSNumber *)bar jsonObject:
NSDicationary *)jsonObject) {
  // ...
}
```

#### 1. Android

将JSON对象传给原生模块的方法时，它们在桥接层被序列化，到了原生层再进行反序列化。原生层会创建`ReadableMap`类型的数据，这种类型与普通的`Map`类似，只不过拥有类型化的API接口（比如`getString`、`getDouble`等）。它同样为JavaScript数组提供了`ReadableArray`接口。

带有`@ReactMethod`注释的方法支持以下参数类型。

```
Boolean -> Bool
Integar -> Number
Double -> Number
Float -> Number
String -> String
Callback -> function
ReadableMap -> Object
ReadableArray -> Array
```

### 2. iOS

iOS平台上JSON对象传到原生层会被转换成普通的NSDictionary类，JavaScript数组则会被转换成NSArray类。RCT_EXPORT_METHOD宏支持以下参数类型。

```
NSString -> String
NSIntegar -> Number
NSNumber -> Number
CGFloat -> Number
float -> Number
double -> Number
BOOL -> boolean
RCTResponseSenderBlock -> function
NSDictionary -> Object
NSArray -> Array
```

## 2.4.3　回调函数和 promise

由于所有的通信过程都是异步的，原生模块不能有返回值。React Native使用回调函数和promise类作为解决方案。使用方式与在JavaScript代码中执行回调函数以及promise的resolve/reject操作完全一样。一系列的参数既可以用来调用回调函数，也可以让promise对象进入成功（resolved）或失败（rejected）状态。只要把回调函数或者promise作为原生模块方法的最终参数，就可以直接使用了，React Native会完成剩下的工作。

### 1. 回调函数

原生回调函数接口遵循两个参数的约定：第一参数表示错误对象（没有错误的情况则为null），第二参数用来提供要响应的数据。

- **Android**

```
@ReactMethod
public void add (int a, int b, Callback callback) {
  int sum = a + b;
  callback.invoke(null, sum);
}
```

- **iOS**

```
RCT_EXPORT_METHOD(add:(NSNumber *)a b(NSNumber *)b callback(RCTResponseSenderBlock)
callback) {
  NSNumber* sum = a + b;
  callback.invoke(@[[NSNull null], sum]);
}
```

值得注意的是iOS在实现方式上的差别。回调函数的接口接受一个数组参数。该数组的内容才是要用在JavaScript回调函数上的真正参数。

- **JavaScript**

```
import {NativeModules} from 'react-native';
const ExampleModule = {NativeModules};

ExampleModule.add(1, 2, function (error, sum) {
  if (error) {
    console.log('An exception occured', error);
  } else {
    console.log(sum) // 3!
  }
});
```

### 2. promise

promise可以在操作成功后把值返回给JavaScript层。两个平台对它的实现有明显的不同，然而思想是一致的。响应要么成功要么失败。JavaScript层的接口保留了我们所熟悉的`.then`和`.catch`方法。

- **Android**

Android的promise接口暴露了`resolve`和`reject`方法。如果promise变成失败（rejected）状态，那么必须提供`Throwable`类的对象。

```
@ReactMethod
public void failIfFalse (boolean value, Promise promise) {
  if (value) {
    promise.resolve("Your value was true, so it resolved.");
  } else {
    promise.reject(new Error("Your value was false, so it rejected"));
  }
}
```

- **iOS**

iOS的promise同时拥有`resolver`和`rejecter`方法。原生模块没有采取包含两个方法的对象形式，而是定义了两个参数。定义接收promise的方法时使用与`RCT_EXPORT_METHOD`不同的宏：`RCT_REMAP_METHOD`。

```
RCT_REMAP_METHOD(failIfFalse:(BOOL *) value,
                 resolver:(RCTPromiseResolveBlock)resolve
                 rejecter:(RCTPromiseRejectBlock)reject)
{
  if (value) {
    resolve(@"Your value was true, so it resolved");
  } else {
    reject(@"Your value was false, so ti rejected");
  }
}
```

- **JavaScript**

```
import {NativeModules} from 'react-native';
const ExampleModule = {NativeModules};

ExampleModule.failIfFalse(true)
  .then(() => console.log('passed!')) //  如果一切没有问题，那就通过
  .catch(() => console.log('failed!'));
```

### 3. 返回请求成功的对象

我们已经了解到，可以通过回调函数和promise在请求成功后把标量值返回给JavaScript层。大多数情况下响应内容应该是JSON对象的形式。

**注意：可以返回枚举类型和类，不过你需要实现一个“转换器”。关于“转换器”的讨论将超出本章节的范畴，不过你可以在React Native文档中有关原生模块的部分去查看它的细节。**

- **Android**

Android提供了`WritableMap`来返回JSON响应。该接口和包含特定类型`put`方法的`Map`很相似。

- ❑ `putNull`
- ❑ `putBoolean`
- ❑ `putDouble`
- ❑ `putInt`
- ❑ `putString`
- ❑ `putArray`
- ❑ `putMap`

`String`类型的键名作为每个方法的第一参数。第二参数是方法名相关类型的值。

```
@ReactMethod
public void jsonResponse(Promise promise) {
  WritableMap jsonMap = new WritableNativeMap();
  jsonMap.putString("stringValue", "A string");
  jsonMap.putBoolean("booleanValue", true);
  jsonMap.putInt("intValue", 123);
  promise.resolve(jsonMap);
}
```

- **iOS**

返回JSON对象和接收它时的情况相似，会创建一个`NSDictionary`类。这个过程非常直白，不会有什么弯路。

```
RCT_REMAP_METHOD(jsonResponse,
                 resolver:(RCTPromiseResolveBlock)resolve
                 rejecter:(RCTPromiseRejectBlock)reject)
{
  resolve(@{
```

```
            @"stringValue": @"A string",
            @"booleanValue": true,
            @"intValue": 123
          });
}
```

## 2.4.4　常量

原生模块也可以导出常量，供JavaScript访问。

```
import {NativeModules} from 'react-native';
const ExampleModule = {NativeModules};

console.log(ExampleModule.HELLO) // world
console.log(ExampleModule.FOO) // foo
```

JavaScript模块访问常量的过程是同步的，并且会把它们作为自己的属性。只要你写下了第一条`NativeModules`的`import`声明，常量属性在应用的整个生命周期都可用。除了常量以外的任何变量都应该通过模块的方法来访问。可以把它们看作传统的静态常量。

要使原生模块的常量可用，需要实现父类的`getConstants`（Android）或`constantsToExport`（iOS）方法，这两个方法将返回键值对的映射，键名会作为JavaScript模块的属性而存在。

### 1. Android

实现`getConstants`方法用于返回包含要导出的常量的`Map`。

```
@Override
public Map<String, Object> getConstants() {
  final Map<String, Object> constants = new HashMap<>();
  constants.put("HELLO", "world");
  constants.put("FOO", "bar");
}
```

### 2. iOS

实现`constantsToExport`方法用于返回包含要导出的常量的`NSDictionary`。

```
- (NSDictionary *)constantsToExport
{
  return @{
            @"HELLO": @"world",
            @"FOO": "bar"
          };
}
```

## 2.4.5　事件

`DeviceEventEmitter`模块用于把事件从原生层传到JavaScript层。

```
import {DeviceEventEmitter} from 'react-native';
```

```
DeviceEventEmitter.addListener('customEvent', e => {
  console.log('received event', e.nativeEvent.param);
});
```

两个平台依然遵循相同的思想，但分别提供了不同的实现细节。

### 1. Android

ReactContext暴露出getJSModule方法用于获取其他模块，为它传入类参数后返回相应的实例。它将用来获取RCTDeviceEventEmitter模块并调用emit方法，接着事件就会通过桥接层发送到JavaScript层，DeviceEventEmitter模块的监听器就会捕获这个事件，并通知其他任意的事件监听器。

用WritableMap类型创建事件对象。

```
@ReactMethod
public void sendTestEvent() {
  String eventName = "customEvent";
  WritableMap params = new WritableNativeMap();
  params.putString("param", "Hello world");

  getReactApplicationContext()
    .getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
    .emit(eventName, params);
}
```

### 2. iOS

在iOS上直接通过对桥接层的引用触发事件。与Android的代码示例类似，JavaScript层的DeviceEventEmitter会接收这些事件。

创建的事件对象为NSDictionary类。

```
@implementation ExampleModule

@synthesize bridge = _bridge;

- (void) sendTestEvent
{
  NSString *eventName = @"customEvent"
  NSDictionary *params = @{@"param": @"Hello world"};

  [self.bridge.eventDispatcher sendAppEventWithName:eventName
                                               body:params];
}
```

### 3. JavaScript

为了让这个示例更加完整，来看一下所有部分的整合。当调用了原生模块的sendTestEvent方法后，应该会接收到来自DeviceEventEmitter模块的事件。

```
import {DeviceEventEmitter, NativeModules} from 'react-native';
const {ExampleModule} = NativeModules;
```

```
DeviceEventEmitter.addListener('customEvent', e => {
  console.log('received event', e.nativeEvent.param); // 2
});

ExampleModules.sendTestEvent(); // 1
```

## 2.5 示例

接下来看一个完整的原生模块实现。我们将创建一个模块用于获取设备电量水平以及状态数据，并且在数据发生变化时可以接收到事件。我们还将把它与React组件绑定，以便在应用中显示一个电池图标。这个示例将阐明原生模块如何直接连接Android和iOS的SDK。该原生模块的代码可以在网页https://github.com/robinpowered/react-native-device-battery上找到。



模块暴露的方法可以用于获取电量水平以及状态数据，并且与`DeviceEventEmitter`进行绑定后可以接收电量数据发生改变的事件。

```
DeviceBattery.isCharging().then(isCharing => {
  // 布尔值isCharging指示设备是否正在充电或者已耗尽电量
});

DeviceBattery.getBatteryLevel().then(level => {
  // level值处于0和1之间，表示电量百分比
});

DeviceEventEmitter.addListener('batteryChange', event => {
  // event.charging包含布尔值
  // event.level包含小数值
});
```



### 2.5.1 Android

为了访问Android设备的电池信息，我们从模块内部直接调用Android SDK。`BroadcastReceiver`和`FilteredIntent`这两个类用来接收电量水平和充电状态改变的信息。模块初始化时会注册receiver以及过滤后的intent，并暴露出两个带有`@ReactMethods`的方法，`getBatteryLevel`和`isCharging`。JavaScript调用了这些方法后，从`FilteredIntent`中提取电量水平以及状态。电池数据也就从原生层流向JavaScript层。一旦电量水平或者充电状态发生改变，就向JavaScript层发出消息，传递包含电池数据的JSON对象。

```
class DeviceBatteryModule extends ReactContextBaseJavaModule
  implements LifecycleEventListener {

  public static final String REACT_MODULE = "DeviceBattery";
  public static final String EVENT_NAME = "batteryChange";
  public static final String IS_CHARGING_KEY = "charging";
  public static final String BATTERY_LEVEL_KEY = "level";
```

```
private Intent batteryStatusIntent = null;
private @Nullable PowerConnectionReceiver batteryStateReceiver;

public DeviceBatteryModule(ReactApplicationContext reactApplicationContext) {
  super(reactApplicationContext);
}

@Override
public String getName() {
  return REACT_MODULE;
}

@Override
public void initialize() {
  // 一般来说最好当模块执行到生命周期的initialize部分时，设置receiver
  // 在ReactNative生命周期中，在构造函数中这样做可能太早了（在应用就绪之前）
  super.initialize();
  getReactApplicationContext().addLifecycleEventListener(this);
  maybeRegisterReceiver();
}

/**
 * 获取电量水平的React方法
 * 通过promise返回数据
 */
@ReactMethod
public void getBatteryLevel(Promise promise) {
  if (batteryStatusIntent != null) {
    //把电量水平返回给JS层
    promise.resolve((double) getBatteryPercentageFromIntent(batteryStatusIntent));
  } else {
    promise.reject(new Error("BatteryManager is not active"));
  }
}
/**
 * 获取电池充电状态的React方法
 * 通过promise返回数据
 */
@ReactMethod
public void isCharging(Promise promise) {
  if (batteryStatusIntent != null) {
    // 把充电状态返回给JS层
    promise.resolve(getIsChargingFromIntent(batteryStatusIntent));
  } else {
    promise.reject(new Error("BatteryManager is not active"))
  }
}
/**
 * 电量水平和充电状态发生改变时被调用
 * 电量水平和充电状态会被传播给JavaScript层
 */
protected void notifyBatteryStateChanged(Intent intent) {
  // 如果访问不到桥接层，不要尝试传播事件
  // 这将会导致“没有模块正在运行”异常（无法访问RCTDeviceEventEmitter）
```

```
  if (getReactApplicationContext().hasActiveCatalystInstance()) {
    // 创建包含电池数据的事件载荷对象
    WritableMap params = getJSMap(intent);

    //当电池状态发生改变时，通知JS层
    getReactApplicationContext()
      .getJSModule(DeviceEventManagerModule.RCTDeviceEventEmitter.class)
      .emit(EVENT_NAME, params);
  }
}

@Override
public void onHostResume() {
  // 应用恢复运行时创建receiver
  maybeRegisterReceiver();
}

@Override
public void onHostPause() {
  // 应用暂停运行时拆解receiver
  maybeUnregisterReceiver();
}

@Override
public void onHostDestroy() {
  //应用停止运行时拆解receiver
  maybeUnregisterReceiver();
}

/**
 * 如果尚未注册receiver，进行注册
 */
private void maybeRegisterReceiver() {
  if (batteryStateReceiver != null) {
    return;
  }
  batteryStateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      //当电池状态发生改变时，通知JS层
      notifyBatteryStateChanged(intent);
    }
  }
  IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
  batteryStatusIntent = getReactApplicationContext()
    .registerReceiver(batteryStateReceiver, filter);
}

/**
 * 如果存在receiver，注销它
 */
private void maybeUnregisterReceiver() {
  if (batteryStateReceiver != null) {
    return;
```

```
    }
    getReactApplicationContext().unregisterReceiver(batteryStateReceiver);
    batteryStateReceiver = null;
    batteryStatusIntent = null;
  }

  /**
   * 从intent中提取电量水平数据
   */
  private float getBatteryPercentageFromIntent(Intent intent) {
    int level = intent.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
    int scale = intent.getIntExtra(BatteryManager.EXTRA_SCALE, -1);
    return level / (float) scale;
  }

  /**
   * 从intent中提取充电状态数据
   */
  private boolean getIsChargingFromIntent(Intent intent) {
    int status = intent.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
    return (
      status == BatteryManager.BATTERY_STATUS_CHARGING ||
      status == BatteryManager.BATTERY_STSTUS_FULL
    );
  }

  /**
   * 构造包含电池数据的WritableMap（JSON载荷）
   */
  private WritableMap getJSMap(Intent intent) {
    WritableMap map = new WritableNativeMap();
    map.putDouble(IS_CHARGING_KEY, getIsChargingFromIntent(intent));
    map.putDouble(BATTERY_LEVEL_KEY, getBatteryPercentageFromIntent(intent));
    return map;
  }
}
```

需要重点关注的方法有以下三个。

- getBatteryLevel
- isCharging
- notifyBatteryStateChanged

以上都是JavaScript层的接口，直接访问Android的原生部分。当getBatteryLevel或isCharging方法被调用后，它们会从包含电池数据的intent中提取数据。当电池状态发生改变时，notifyBatteryStateChanged方法被触发，并把电池数据传给JavaScript层。

这个例子很好地展示了原生模块的紧密结合，使得原本看似受限或不可访问的Android SDK可以被JavaScript直接访问。本模块的代码既不复杂也不庞大，大部分内容都是关于恰当地创建和拆解BroadcastReceiver，其余的部分仅仅是将其与ReactMethods还有事件触发器捆绑到一起。

## 2.5.2 iOS

iOS的原生模块暴露了同样的接口和能力，定义了isCharging和getBatteryLevel方法，两者都返回promise，以及电池状态变化的处理函数，通过DeviceEventEmitter传播电池状态。iOS通过UIDevice访问电量水平以及充电状态数据，并通过NSNotificationCenter来订阅电池状态变化的消息。

```
// DeviceBattery.h
#import <Foundation/Foundation.h>
#import "RCTBridgeModule.h"
#import "RCTBridge.h"
#import "RCTEventDispatcher.h"

@interface DeviceBattery : NSObject<RCTBridgeModule>
@end

// DeviceBattery.m
#import "DeviceBattery.h"
#import "RCTBridge.h"
#import "RCTEventDispatcher.h"

@implementation DeviceBattery
@synthesize bridge = _bridge;

RCT_EXPORT_MODULE();

-(instancetype)init
{

  if((self = [super init])) {
    [[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];

    // 订阅电池状态改变的消息
    [[NSNotificationCenter defaultCenter] addObserver:self
                                              selctor:@selector(batteryLevelChanged:)
                                                 name:UIDeviceBatteryLevel-
DidChangeNotification
                                               object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                             selector:@selector(batteryLevelChanged:)
                                                 name:UIDeviceBatteryState-
DidChangeNotification
                                               object:nil];
  }
  return self;
}

RCT_REMAP_METHOD(isCharging,
                 isChargingResolver:(RCTPromiseResolveBlock)resolve
                 isChargingRejecter:(RCTPromiseRejectBlock)reject)
{
  UIDeviceBatteryState batteryState = [UIDevice currentDevice].batteryState;
```

```
  // 把充电状态数据返回给JS层
  if (batteryState == UIDeviceBatteryStateCharging) {
    resolve(@YES);
  } else {
    resolve(@NO);
  }
}

RCT_REMAP_METHOD(getBatteryLevel,
                 batteryLevelResolver:(RCTPromiseResolveBlock)resolve
                 batteryLevelRejecter:(RCTPromiseRejectBlock)reject)
{
  // 把电量水平数据返回JS层
  float batteryLevel = [UIDevice currentDevice].batteryLevel;
  resolve(@(batteryLevel));
}

-(void)batteryLevelChanged:(NSNotification *)notification
{
  NSMutableDictionary* payload = [NSMutableDictionary dictionaryWithCapacity: 2];

  // 收集充电状态和电量水平数据
  bool isCharging = [UIDevice currentDevice].batteryState == UIDeviceBatteryStateCharging;
  float batteryLevel = [UIDevice currentDevice].level;

  // 把包含充电状态和电量水平数据的事件传播给原生层
  [payload setObject:[NSNumber numberWithBool:isCharging] forKey:@"charging"];
  [payload setObject:[NSNumber numberWithFloat:batteryLevel] forKey:@"level"];
  [self.bridge.eventDispatcher sendDeviceEventWithName:@"batteryChange" body:payload];
  }

-(void)delloc
{
  // 模块销毁后拆解观察者（observer）
  [[NSNotificationCenter defaultCenter] removeObserver:self];
}

@end
```

`init`代码块中创建了观察电池状态变化的观察者（observer）。如果充电状态或者电量水平发生变化，观察者会调用`batteryLevelChanged`方法。该方法内访问了包含电池充电状态和电量水平的`[UIDevice currentDevice]`。接着生成事件对象并分发到JavaScript层。

JavaScript调用另外两个方法完成同样的任务，访问`UIDevice`的电池属性并用promise返回数据。

## 2.5.3　JavaScript

接下来把刚刚编写的原生模块放到真实的示例中使用。我们想要显示一个电池条来指示当前的电量水平以及充电状态。电池条将显示三种颜色来表示不同的电池状态：红色表示电量低，绿

色表示正在充电，其他情况下显示为白色。电池充电的过程中，电池条会逐渐填满。

```
import React from 'react';
import {NativeModules} from 'react-native';
const {DeviceBattery} = NativeModules;

class BatteryIcon extends React.Component {
  constructor(props) {
    super(props);

    // 这里初始化默认状态，等组件挂载时立刻更新
    // 也可以在此调用原生模块
    this.state = {
      battery: {level: 0, charging: false}
    };
  }
  componentWillMount() {
    // 订阅电量水平/充电状态的改动
    this.batteryListener = DeviceBattery.addListener(
      'batteryChange',
      ({charging, level}) => this.setState({charging, level})
    );

    // 当组件挂载后，获取电量水平和充电状态
    DeviceBattery.isCharging().then(charging => this.setState({charging}));
    DeviceBattery.getBatteryLevel().then(level => this.setState({level}));
  }
  componentWillUnmount() {
    // 总是在组件即将卸载时移除所有监听器
    this.batteryListener.remove();
  }
  render() {
    const {charging, level} = this.state;
    return (
      <View style={styles.container}>
        {/*显示电量水平*/}
        <Text>{Math.round(level * 100)}%</Text>
        {/*显示成电池条的形式*/}
        <PowerBar charging={charging} level={level} />
      </View>
    )
  }
}

function PowerBar ({charging, level}) {
  // 电池条填充的宽度，以百分之一为单位
  const fillPercentage = level / 100;
  // 电池条的背景色
  // 电量<20为红色
  // 充电为绿色
  // 未充电且电量>20为白色
  const backgroundColor = level <= 20 ? 'red' : charging ? 'green' : 'white';
  return (
    <View style={styles.batteryContainer}>
```

```
      <View style={[styles.batteryBar, {flex: fillPercentage, backgroundColor}]} />
      <View style={{flex: 1 - fillPercentage}} />
    </View>
  );
}
```

从react-native中导入NativeModules，并通过它的属性访问DeviceBattery模块。在组件挂载时添加一个DeviceBattery模块的监听器，用于监测任何的数据变动。之前我们已经编写好代码让原生模块传播充电状态和电量水平数据。一旦接收到数据，就根据它们来更新组件状态。

第一次显示组件时，由于事件不会立刻传播过来，我们要执行初始的getBatteryLevel和isCharging请求来获取数据。这两个方法通过promise返回的结果会被写到状态中。

此刻我们已经从原生模块获取了一切所需的数据，接下来就可以把它们展示出来了。

### 2.5.4　注意事项：线程

原生模块不应该关心哪个线程调用了它。React Native还处于逐渐成熟的过程，无法保证它的机制（也就是内部的实现细节）在未来会不会保持现状。

回想一下桥接层在两层之间**批量**执行异步通信的过程。如果一个原生模块有阻塞请求的可能，就应该延缓繁重的操作并把它们交给内部管理的工作进程。一条阻塞的请求会延迟同一批处理队列中其他请求的执行。

在iOS系统中，如果某个方法需要在主线程上执行，原生模块可以像下面这样写。

```
-(dispatch_queue_t)methodQueue
{
  return dispatch_get_main_queue();
}
```

类似地，如果某些方法会造成阻塞请求的代价，它可以创建自己的队列。下面的代码以RCTAsyncLocalStorage为例。

```
-(dispatch_queue_t)methodQueue
{
  return dispatch_queue_create("com.facebook.React.AsyncLocalStorageQueue",
DISPATCH_QUEUE_SERIAL);
}
```

值得注意的是，methodQueue将会指定模块中所有方法的行为。dispatch_async可以用来在独立队列中执行任务。

```
RCT_EXPORT_METHOD(doSomethignExpensive:(NSString *)param callback:(RCTRespon-
seSenderBlock)callback)
{
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
  ^{
    // 在后台线程执行任务
```

```
    ...
    // 回调函数可以被任意线程或队列调用
    callback(@[...]);
  })
}
```

在Android平台上可以新建一个Runnable，当需要返回数据时，可以在Runnable内部安全地调用回调函数或者promise。

### 2.5.5 注意事项：Swift

原生模块可以用Swift来编写，不过需要在Objective-C宏的帮助下把方法注册到桥接层。



把ExampleModule的代码写成Swift类，如下所示。

```
// ExampleModule.swift

@objc(ExampleModule)
class ExampleModule : NSObject {

  @objc func callableMethod(foo: String, bar: String) -> Void {
    ...
  }
}
```

一个私有文件将用来把模块和模块方法注册到桥接层。RCT_EXTERN_MODULE和RCT_EXTERN_METHOD宏会创建映射关系与Swift类的实现进行关联。

```
// ExampleModule.m
#import "RCTBridgeModule.h"

@interface RCT_EXTERN_MODULE(ExampleModule, NSObject)
RCT_EXTERN_METHOD(callableMethod:(NSString *)foo bar:(NSString *)bar)

@end
```

最后一点，在一个iOS项目中用两种语言混合开发时，总是需要引入桥接文件。

```
// ExampleModule-Bridging-Header.h
#import "RCTBridgeModule.h"
```

## 2.6 链接模块和组件

本节内容只针对Android平台。

在Android平台上，所有自定义组件和模块都要用ReactActivity进行注册，以便React Native找到它们。ReactActivity的getPackages方法在MainActivity中实现。默认情况下，它返回一个列表，并以MainReactPackage作为唯一的入口。这个包内含有React Native自带的所有核心原生模块和组件。Android平台的实现很优雅，考虑到了原生模块和组件层的核心activity以及桥接层的解

耦。所有模块和组件都在ReactPackage中注入到整个项目中。

在iOS平台上，整个应用都可以通过RCT_EXPORT_MODULE宏来访问所有模块和组件。

ReactPackage接口提供了三个方法。

- createNativeModules——包含所有自定义的原生模块。
- createJSModules——没有使用过而且没有文档说明，不过可以用它把JavaScript模块注册到Catalyst实例上，但是无法保证其会成为主文件包的一部分。
- createViewManagers——包含所有自定义的原生视图管理器。

我们将用到前文编写的示例，把组件和模块添加到新的项目中。DeviceBattery作为自定义模块，ReactMapBoxViewManager作为自定义UI组件，我们要为它提供支持。

创建自定义包文件。

```
public class YourAppPackage implements ReactPackage {
  @Override
  protected List<NativeModule> createNativeModules(ReactApplicationContext
reactApplicationContext) {
    return Arrays.<NativeModule>asList(
      new DeviceBattery(reactApplicationContext) // 自定义模块
    );
  }

  @Override
  public List<Class<? extends JavaScriptModule>> createJSModules() {
    return Collections.emptyList();
  }

  @Override
  public List<ViewManager> createViewManagers(ReactApplicationContext
reactApplicationContext) {
    return Arrays.<ViewManager>asList(
      new ReactMapBoxViewManager() // 自定义视图管理器
    );
  }
}
```

这个类接下来会在MainActivity中作为getPackages的一部分被实例化。

```
// MainActivity.java
class MainActivity extends ReactActivity {
  // ...

  @Override
  protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
      new MainReactPackage(), // 默认情况下，包含所有核心模块和组件
      new ExamplePackage() // 所有自定义模块和组件
    );
  }
}
```

现在ReactPackage已经注册到React Native中了，应用的JavaScript层和原生层都可以访问自定义的模块和组件。

## 链接第三方库

React Native的模块和组件可以写成项目外部依赖的形式，而不用创建整个React Native项目。第三方React Native模块的安装方式和一般的Node模块一样，通过`npm install`命令安装。把它们正确地链接到原生层之前，还有几步操作要手动完成。因此出现了React Native包管理器rnpm来自动执行这个链接过程。

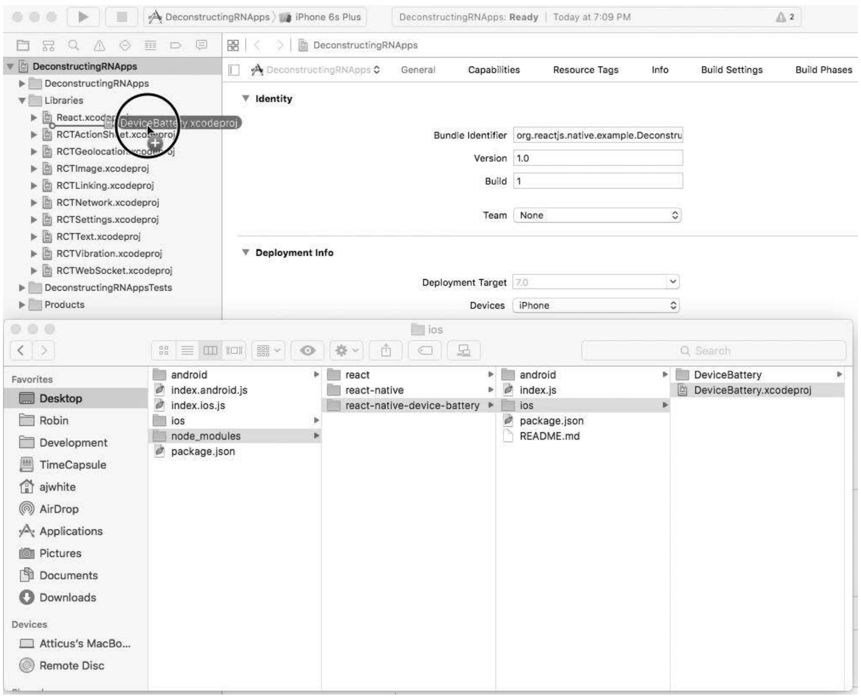


先来看一下手动链接过程。我们想要安装`react-native-device-battery`模块，也就是之前用来访问电池电量和状态的原生模块。

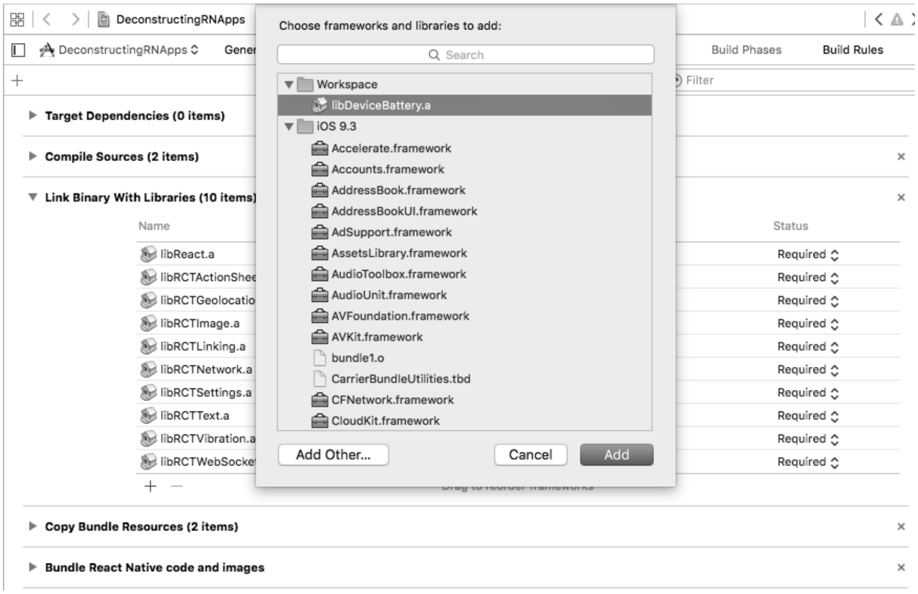
第一步，运行`npm install react-native-device-battery --save`以将模块添加到项目中。这步操作将会把依赖安装到node_modules/react-native-device-battery目录下。

### 1. iOS

用XCode打开iOS项目，同时打开Finder窗口并前往项目的node_modules/react-native-device-battery目录，在该目录下找到ios/目录下的DeviceBattery.xcodeproj文件，把它拖到XCode文件树的Libraries分组中。



下一步，切换到Build Settings标签，展开Link Binary With Libraries分组，点击加号按钮。接着会弹出所有可供链接的二进制文件列表，列表顶部的Workspace文件夹下有一个DeviceBattery.a文件，选择并把它添加到已链接的二进制文件中。




此时一切文件都正确地链接完毕，可以编译项目了。链接模块已经就绪，可供使用，除非某个依赖项的库需要在`AppDelegate`中进行初始化。

### 2. Android

在Android平台上我们主要对gradle文件进行编辑，并把依赖添加到`MainActivity`的`ReactPackages`列表中。

首先打开android/settings.gradle设置文件，添加以下代码。

```
rootProject.name = 'NameOfYourProject'

include ':app'
+include ':react-native-device-battery'
+project(':react-native-device-battery').projectDir = new File(
+  rootDir, '../node_modules/react-native-device-battery')
```

接着在构建文件android/app/build.gradle中添加一行代码。

**注意：有两个build.gradle文件，一个是android/build.gradle，而要编辑的那个是android/app.build.gradle。**

如下所示，`dependencies`部分添加了一行代码，该部分通常位于文件最后。

```
dependencies {
  compile fileTree(dir: "libs", include: ["*.jar"])
  compile "com.android.support:appcompat-v7:23.0.1"
  compile "com.facebook.react:react-native:+"
+ compile project(':react-native-device-battery')
}
```

最后，打开MainActivity.java，把依赖项的ReactPackage添加到getPackages方法返回的集合中。

```
// MainActivity.java

+import com.robinpowered.react.DeviceBatteryPackage;

class MainActivity extends ReactActivity {
  // ...

  @Override
  protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
      new MainReactPackage(),
+     new DeviceBatteryPackage()
    );
  }
}
```

现在整个应用都可以访问刚刚添加的模块了。

### 3. rnpm

如前文所示，链接React Native依赖的过程需要相当多的手动操作。我们要把这种做法抛之脑后，转而使用一个新的工具rnpm。rnpm的使命就是简化依赖链接的过程，避免什么事都要手动完成。

> React Native包管理器的使命在于简化React Native的日常开发过程。它受CocoaPods启发，作为你进行react-native链接的最佳伙伴，引导你解决原生层中不熟悉的部分。它致力于在不需要任何额外配置的情况下，就能用几乎所有的第三方包进行开发。

rnpm作为全局的Node模块，可全局运行`npm install`来安装它。

```
npm install rnpm -g
```

- **安装依赖**

如果你想要安装某个依赖并进行链接，在项目根目录下运行以下命令。

```
rnpm install <name>
```

- **链接依赖**

如果你已经安装（未进行链接）某些依赖，在项目根目录下运行以下命令。

```
rnpm link
```

当然，如果只有一个依赖需要链接，在上述命令后跟上依赖包名即可。

```
rnpm link <name>
```

## 2.7　总结

本章深入研究了React Native的组件和模块，讲解了它们的工作原理，如何编写，以及如何将其整合到React Native项目中。你要掌握的便是原生组件以及模块的开发，模块其实仅仅是一个类，作为你所要用到的原生功能或UI组件的代理层。原生模块就是JavaScript层可以调用的一些方法的集合。React Native做了大量的工作，将JavaScript层与原生层连接起来。原生的组件管理器也仅仅是一些方法的集合，当设置或者改变JavaScript组件属性的时候，可以调用这些方法。而上述两者就是原生层的入口。React Native已经为你解决了所有的难题。

第 3 章

# 示例应用：Myagi

Myagi成立于2013年，为零售业销售人员提供在线培训平台。Myagi开发的目的在于让零售业员工可以更便捷地获取工作所需的信息。没有Myagi的情况下，零售商几乎无法为员工提供连续且可跟踪的培训体验。而有了Myagi的帮助，他们就能为员工提供视频形式的培训资料，员工通过这些资料可以了解到工作以及所售产品的重要方面。这些培训单元能确保为员工连续提供信息，同时管理者可以更方便地跟踪哪个员工接受了哪种练习。

提供这样一个平台的重点在于，允许用户在移动端访问培训资料。尽管已经有了专为移动端优化的Web应用，但显而易见的是，原生应用能够极大地提升移动端的用户体验。移动Web应用无法提供原生应用那样的性能以及触摸交互体验。因此，我们决定用React Native开发移动应用。你可以前往网页https://itunes.apple.com/us/app/myagi/id1093253823下载应用，看看我们的最终成果。



## 3.1　为什么选择 React Native

Myagi的团队相对较小，因此在开发原生应用的同时维护Web应用（全球每天都有许多用户在使用）将十分困难。然而，我们的Web应用是用React开发的，所以我们希望使用React Native可以复用Web应用的部分代码以及开发技巧。

尽管React Native似乎是很好的选择，我们还是考虑了两个替代方案：完全用Swift开发iOS应用，或者用Phonegap这样的Web包装器来开发。

第一个“完全原生”的方案很吸引人，因为它让我们确信：如果采用Swift，那么可以开发出界面和功能与市场上的顶尖iOS应用一样棒的高质量应用。然而，这需要学习一套全新的技术集，并且无法复用已有的前端代码。因此我们放弃了这个方案。

第二个替代方案便是用Phonegap打包Web应用，让它可以从应用商店中下载。用Phonegap（http://phonegap.com/）“打包”一个网站十分简单，这样就能在iOS和Android的应用商店中下载到应用形式的网站。这个方案的优点在于，我们基本上只需要把已有的Web应用打包好，就几乎可以马上为用户提供一个应用。然而我们否决了这个方案，因为Phonegap应用的用户体验不如原

生应用。

因此，我们选择了React Native。它允许我们提供完美的用户体验，并且能够复用现有的代码和技巧。这种在原生应用和Web应用间共享现有代码的能力，极大地减少了花在开发原生应用上的时间。后续章节中会详细描述实现过程。

### 开始前的准备

本章将通篇介绍开发React Native应用的过程中如何解决一些关键的技术难题，我们只有iOS平台的应用，因此我的一些建议只针对iOS平台，不过大部分建议都适用于两个平台。总的来说，我希望接下来要介绍的解决方案能够帮助其他人更容易解决遇到的类似问题。

## 3.2　状态

状态管理是任何前端应用中最困难的部分之一（不论React Native应用还是其他应用）。可以肯定，你几乎需要不断地通过API来获取和更新应用的数据，除非你的应用通过本地存储来持久化所有状态。这会带来下面的一系列挑战。

- 确保组件有权限访问它们所需的数据→不同组件对数据的需求有很大的不同。比如，某个组件可能仅仅需要获取当前用户的`email`属性，而另一个组件可能既需要`email`属性，又需要用户的`company_name`属性。确保两者都能得到它们所需的数据，又不用重复调用API，这将是一个挑战。
- 分页（pagination）→数据量很大的情况下，不可能一次就把数据实体的全部集合都取回来，相反，很有必要对数据实体进行分页。
- 缓存→启用缓存可以极大地提升用户体验。以Myagi为例，当用户第一次打开应用时，获取当前用户的培训计划。当用户下一次访问首页时，没有理由让他们等待重新获取所有培训计划。我们的做法是对数据进行缓存，并重新展示出来。这也是一个很有挑战性的任务，因为需要保证当服务端的培训计划的某个属性发生改变时，该变化要展现于用户眼前（即让缓存失效）。

庆幸的是，着手开发React Native应用之前，在基于React开发的Web应用中，我们已经有了大型的代码库，使得状态的获取和更新变得非常简单。这个状态管理代码方案使用了Marty.js（http://martyjs.org/）以及Flux架构（http://facebook.github.io/flux/docs/overview.html#content），并且我们在此基础上开发了一整套代码方案，使得应用获取和更新状态变得很方便。Myagi应用的这个状态管理库是我们为自己的API量身定做的，另外所用到的Marty.js如今已不再维护，因此没有把它开源。不过我会详细介绍它的工作原理，因为我相信其中的大部分工作也可以适用于其他情况。

### 3.2.1 Flux

在进一步详细介绍应用之前，很有必要先来了解一下Flux。Flux是一种应用架构，可以简化前端应用的状态管理过程。这里做一个简单的解释，Flux包含以下几个关键概念。

- store→用来保存数据。组件可以通过它们获取数据。
- action生成器→用来更新数据以及生成新的数据。
- 分发器（dispatcher）→Flux架构的应用中通常只有一个分发器。一般来说，分发器作为action生成器和store的通信中介。store把回调函数注册到分发器上，当action生成器发送了一条匹配某个回调函数的信息时，就会触发对应的回调函数。

这只是对Flux的简要概述，因此如果想要深入地理解这个架构，就得仔细阅读它的文档。重要的是，Flux仅仅是一种架构，所以每个应用都需要自己来具体实现它。Marty.js库如今已不再维护，它原本的设计目的在于实现Flux架构。尽管我们目前在Myagi应用中使用了Marty.js，如果可以重选一次，我们很可能会用更流行的Redux来取代它。

### 3.2.2 Myagi API

在讲解如何使用Flux架构以及Marty库管理应用状态之前，先来了解一下Myagi API的特性。Myagi的API是RESTful风格的HTTP API，并具有以下几个重要特性。



- 向数据源发起请求时，可以传递`fields`参数，以便有选择地获取字段并展开相关字段。比如，获取ID为123的培训计划以及它所有子模块的名称，可以发起如下请求：`/api/v1/training_plans/123/?fields=name,modules.name`。通过指定字段`name`和`modules. name`，API就知道应该返回以下格式的响应。

```
{
 name: 'Some plan',
 modules: [
  {
   name: 'First module'
  },
  {
   name: 'Second module'
  }
 ]
}
```

该字段的展开方式基本上可以有无限的可能，这就使得API变得很灵活，任何组件都可以指定并接收自己需要的数据格式。

- 任何列表路径（list endpoint）都可以通过传递指定的查询参数进行筛选。比如我们希望获取名称里包含sales的培训计划，可以发起如下请求：`/api/v1/training_plans/?name__contains=sales`。

- 任何列表路径都可以很方便地用`limit`和`offset`参数进行分页。因此假如想要返回第5个之后的10个培训计划，可以发起如下请求：`/api/v1/training_plans/?offset=5&limit=10`。
- 任何列表路径的查询结果都可以通过`ordering`字段进行排序。比如按名称字段的字母表顺序对培训计划进行排序：`/api/v1/training_plans/?ordering=name`。

### 3.2.3　Marty.js 与状态模块的生成

Marty提供了基础的类和方法来创建Flux的关键组件。举例来说，它提供了`Store`和`ActionCreator`类，可以用它们创建特定数据源的store和action生成器。然而，为了简化数据获取和更新的过程，我们有必要为自己的API量身打造一个状态管理库。这样做的目的在于让所有排序、筛选和缓存的逻辑完全由状态模块自动处理，而组件本身不用对这些进行判断。

我们还希望尽可能地简化代码模板，以便减少整个代码库的冗余。大多数Flux架构的应用到后来都会包含大量的代码模板，因为通常情况下每种数据实体类型（如用户、公司、培训计划等）都要用到许多不同的函数和常量，以便可以通过store和action生成器对它们进行访问和更新。尽管Marty.js（以及其他基于Flux的库）已经简化了代码模板，但我们希望能够进一步简化，于是开发了`stateModuleGenerator`方法，它可以根据给出的数据实体名称和API路径，生成对应的常量、store以及action生成器。比如，用它创建`UsersState`模块的方式如下。

```
import stateModuleFactory from 'state/common/factory/http';

let UsersState = stateModuleFactory({
  entity: 'user',
  endpoint: 'api/v1/users'
});

export UsersState;
```

`UsersState`对象拥有`Store`和`ActionCreators`属性，通过前者可以访问用户数据，通过后者可以更新用户数据。

接下来这个状态模块就可以被我们的“容器”组件所使用了，容器组件用于从store中获取数据并立即传递给单一的子组件。Flux架构的库中普遍存在容器组件的概念：Redux、Relay、Marty.js等库都有这样的概念。这些容器用于通过store请求数据，并在数据返回或者更新时把它传递给子组件。在我们的代码库中，负责渲染用户列表的子组件`UserList`，它的容器组件如下。

```
...

// 渲染用户列表的组件
const UserList = ({ users }) => {
 return (
  <View>
   { users.map( user => <Text key={user.get('id')}>{user.get('first_name')}</Text>)
   }
  <View>
```

```
  )
};

// 容器组件
const UserListContainer = Marty.createContainer(UserList, {

 listenTo: [
  UsersState.Store
 ],

 fetch: {
  users: function() {
   return UsersState.Store.getItems({
    // 只获取每个用户的first_name属性
    fields: [
     'first_name'
    ],
    // 只获取'AcmeCorp'公司的用户
    company__company_name='AcmeCorp'
   });
  }
 }

});
```

如代码所示，`UserListComponent`组件利用之前生成的`UsersState`模块从API获取用户数据。重要的是，容器组件仅仅声明了它需要什么数据（所有`'AcmeCorp'`公司的用户，并只带有`'first_name'`属性），store负责决定从本地缓存还是从远端获取数据，亦或两方面的数据都取。这样的处理方式非常有用，因为所有关于数据缓存和分页的复杂逻辑，都交给store来处理了，容器要做的只是请求特定的数据集并等待接收。

现在，你可能对`UsersState.Store`模块的内部原理感到好奇。实际上这并不重要，因为它和我们的API关联得太紧密了，为了复制它的实现而进行一番详细描述没有什么意义。我反而想演示一下自定义状态管理函数以及辅助工具的开发，以有利于你们在已有的Flux库（如Redux）之上进行React Native开发。这么做可以让你在需要管理状态时维护整个代码库的一致与简洁。

## 3.3　路由

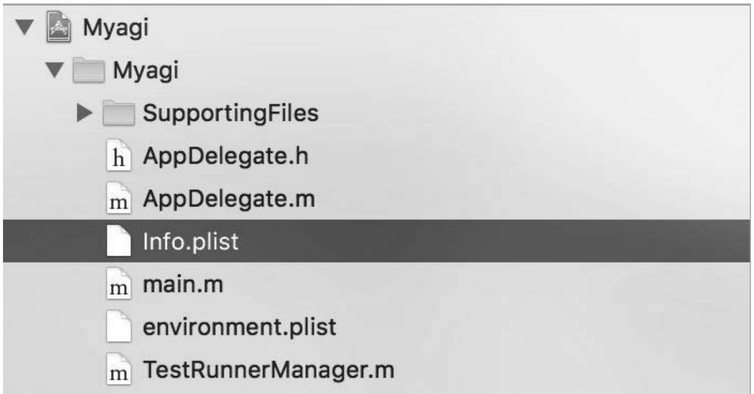
原生应用的路由与Web应用的路由有一些不同。最主要的区别在于原生应用有视图“栈”这一很明显的特点。栈（stack）就是一系列的页面逐个叠放在一起。以此处的应用为例，用户点击了应用中的一个培训计划，一个显示该计划详情的视图就会放置在当前视图的上方。当用户点击了左上方的返回按钮，这个新的视图就会被“弹出”栈，返回到培训计划列表。管理这些栈，以及诸如使用标签页等不同的路由方法，将是一大挑战。我们使用了React Native Router Flux（https://github.com/aksonov/react-native-router-flux）这个库，后续的章节中会介绍它。

## 深度链接

深度链接允许其他应用、网络链接或者通知消息触发启动你的应用，并过渡到某个特定的页面。这种方式很像打开某个网站的URL后自动跳到另一个网站的URL。iOS和Android应用都支持深度链接。以iOS应用为例，你可以为自己的应用注册一个特定的URL Scheme，其他应用就能通过它来启动你的应用。

以此处的应用为例，我们注册了名为myagi的URL scheme。其他应用甚至网站都可以用这个scheme，通过打开myagi://register这样的URL来触发启动我们的应用。访问这个URL后，应用就会启动（如果用户安装了它）并得到所传来的链接。

为你的应用设置这样的URL scheme其实很简单。如果想这么做，只需要选择应用的Info.plist文件。




然后在URL types下的URL Schemes设置中新增一项内容。

| | | |
|---|---|---|
| ▼URL types | Array | (1 item) |
| ▼Item 0 (Editor) | Dictionary | (3 items) |
| Document Role | String | Editor |
| URL identifier | String | com.myagi.app |
| ▼URL Schemes | Array | (1 item) |
| Item 0 | String | myagi |

上述示例中把scheme设置成了myagi，你也可以把它设置成任意值。这里提供一条建议：尽量为你的应用选择独一无二的URL scheme，以免它和用户设备上其他应用的scheme发生冲突。

一旦设置好URL scheme用来启动应用，就必须对传来的链接做某种处理，以便用户能被重定向到正确的页面。以该应用为例，React Native Router Flux库没有内置每个应用内页面的唯一路径这种概念。因此，需要提供自己的解决方案来处理这些深度链接。这里的做法比较直接：当

应用打开后，用React Native提供的Linking模块检查一下是否由深度链接所打开。这个过程由Launch组件完成，无论应用何时启动，加载的第一个页面都是该组件。如果取得了链接，就把它传给handleIncomingLink函数，代码如下。

```
class Launch extends React.Component {
...

  Linking.getInitialURL().then(url => {
    let handled = false;
    if (url) {
      // 该处理函数成功处理了链接之后将返回true
      handled = handleIncomingLink(url);
    }
    if (!handled) {
      // 如果处理函数执行失败或者URL不存在的情况下，过渡到某个默认的路由
      ...
    }
  }
...
}
```

handleIncomingLink函数与其他特定路由的处理函数都是一起在linking模块中进行定义的，而且非常简单。以下是该模块的主要内容。

```
// 处理函数如果成功处理了链接，那就返回true，否则返回false
const APP_LINK_HANDLERS = {
  'register': () => {
    goToPath('unauthedUserStart/register');
    return true;
  }
};

export function handleIncomingLink(url) {
  let alUrl = new AppLinkURL(url);
  let route = alUrl.url.hostname + (alUrl.url.pathname || '');
  let handler = APP_LINK_HANDLERS[route];
  if (handler) return handler();
  return false;
}
```

这段代码中定义了特殊的APP_LINK_HANDLERS对象。如果它的方法与特定的深度链接匹配，就会被调用。比如，我们已经为register路由设置了一个方法，因此如果应用通过myagi://register这一URL来打开，接下来就会用goToPath函数把用户带到注册页面。goToPath函数是问题的最后一环，它在路由工具库中进行定义，参见以下代码。

```
import _ from 'lodash';
import {Actions as RouterActions} from 'react-native-router-flux';

export function goToPath(path) {
  if (!path) return;
  let parts = path.split('/');
```

3

```
  RouterActions[_.head(parts)]();
  _.defer(()=>{
    goToPath(_.tail(parts).join('/'));
  });
}
```

这是一个非常简单的函数，并且它把path参数的各部分高效地处理成独立的路由行为，然后逐个执行。通过这种方式，提供我们定义的RouterActions的正确序列来触发前往对应的路由，并确保为链接设置好处理函数，就可以根据深度链接轻松过渡到应用的任何部分。

这些简单的设置就绪后，就能为应用定义任何想要的自定义链接。尽管目前只有一个myagi://register链接的处理函数，不过添加任何想要的链接也很简单，只要在APP_LINK_HANDLERS对象中定义恰当的处理函数即可。

## 3.4　身份验证

这里选用JSON网络令牌（JWT）来验证应用中的请求。JWT令牌本质上是一串很长且唯一的字符串，它将伴随每条请求一起发送，以便验证它并确保它属于特定的用户（实际上JWT令牌内硬编码了准确的用户信息）。每个JWT令牌用一个密钥进行签名，该密钥只保存在你的服务端（并且不能与任何人共享），因此服务端实际上不需要记录它创建的所有JWT令牌，仅需简单地检查所收到的每个JWT令牌是否用正确的密钥签名即可。如果令牌正确，服务端就可以信任它携带的信息并对请求进行验证。

在Myagi应用中，用户输入登录信息后，一条请求就会发送给后端。如果登录信息正确，后端就会返回一个JWT令牌，接下来就可以用它来为用户获取数据。因此客户端需要保存这个JWT令牌，以便每条请求都能使用它。这就是AuthStore所负责的工作。当取回JWT令牌后，它会马上被传给AuthStore。这样，每当要进行fetch请求时，就会从AuthStore中取出JWT令牌并和请求一起发送。

这个过程中唯一的复杂之处在于，JWT令牌需要持久化保留，甚至应用进程被杀掉（比如设备重启时）也不能丢失。本例为此使用了React Native提供的AsyncStorage模块。auth状态模块的定义见以下代码。

```
import React from 'react-native';
const {
  AsyncStorage
} = React;
import Marty from 'marty-native';
import _ from 'lodash';
import Im from 'immutable';

import app from 'core/application';

const TOKEN_KEY = 'authToken';
```

```
const Constants = Marty.createConstants([
  'SET_AUTH_TOKEN'
]);

// 这些ActionCreators类可以用来设置或清除当前的authToken
class ActionCreators extends Marty.ActionCreators {
  setAuthToken(str) {
    this.dispatch(Constants.SET_AUTH_TOKEN, str);
  }

  clearAuthToken() {
    this.setAuthToken('');
  }
}

// 该store用来记录身份验证令牌
class Store extends Marty.Store {

  constructor(opts) {
    super(opts);
    // 监听action SET_AUTH_TOKEN，并进行相应的处理
    this.handlers = {
      onSetToken: Constants.SET_AUTH_TOKEN
    };
    this.state = {
      authToken: null,
      tokenHasBeenFetchedFromCache: false
    };
    // 尝试从当前存储中为authToken获取初始值
    this.getTokenFromStorage();
  }

  getTokenFromStorage() {
    // 用AsyncStorage模块尝试获取上一次使用应用时保存的令牌
    AsyncStorage.getItem(TOKEN_KEY).then((token)=>{
      // token不存在的情况下值为null，因此不管是否取到值，都对该store的内部状态进行设置
      this.setState({
        authToken: token,
        tokenHasBeenFetchedFromCache: true
      });
    });
  }

  onSetToken(token) {
    // 在当前内存中保存令牌的引用，以便每条请求都可以使用它
    this.state.authToken = token;
    // 持久化保存令牌以便应用重启后可以再次使用它
    AsyncStorage.setItem(TOKEN_KEY, token);
    this.hasChanged();
  }

  getToken() {
    return this.state.authToken;
  }
```

3

```
  tokenHasBeenFetchedFromCache() {
    // 该方法用来检查是否还在等待令牌从持久化存储中取回
    return this.state.tokenHasBeenFetchedFromCache;
  }
}

export default {
  ActionCreators: app.AuthActionCreators,
  Store: app.AuthStore,
};
```

该store的`getToken`方法在每次向服务端发起请求时被调用，用于发送正确的JWT令牌。我们只需确保用action生成器`setAuthToken`在某个时刻设置好令牌。这个过程发生在登录请求成功后的回调函数中。以下是登录页组件的部分代码，它展示了发起用户登录请求并在登录成功后保存返回的令牌。

```
...

// 该action生成器会向'token_auth'路径发起请求
PublicUsersState.ActionCreators.doListAction(
   'token_auth',
    // data参数包含了用户输入的邮件地址和密码
   data
 ).then((res)=>{
   // 此处我们使用了action生成器setAuthToken来保存服务端返回的令牌
   // 该令牌可以被后续的全部请求所用
   AuthState.ActionCreators.setAuthToken(res.body.token);
   ...
}).catch( err => {
   ...
});

...
```

一旦这条保存身份验证令牌的请求完成后，后续的全部请求都可以使用这个令牌。然而，如果用户彻底关闭应用再重启，我们不希望他们需要再次登录。这正是用`AsyncStorage`模块把`authToken`保存在持久化存储中的原因。当启动应用时，如果从`AsynStorage`中取到了`authToken`，就能直接跳过登录页。不过这里会有些复杂，因为很显然`AsyncStorage`是异步的，这意味着一定要等待它检查完令牌是否已经存在，再选择要把用户带到哪（登录或培训页面）。

这个问题的解决方案就是使用之前提到过的专门的`Launch`组件，它在应用启动时被加载为第一个页面。`Launch`组件通过一个容器高效地封装起来，容器负责监听之前所定义的`AuthState.Store`的变化。`Launch`组件本身很简单，渲染一个空的视图并通过执行环境传入`isAuth`值。当`AuthState.Store`还在本地存储中查找已有的令牌时，`isAuth`的初始值为`null`。如果找到了已有的令牌，`isAuth`将被设置为`true`，否则设为`false`。一旦显式地设置了`true`或者`false`，`Launch`组件就能过渡到正确的页面。相关的逻辑写在`Launch`组件的`componentWillUpdate`方法中，如下所示。

```
componentWillUpdate(nextProps, nextState, nextContext) {
  ...
  // 一旦isAuth被设置了null以外的值，就能决定先带领用户前往哪个路由
  if (nextContext.isAuth !== null) {
  ...
    // 如果还未设置currentUser，那么存在身份验证令牌（也可以在此处检查isAuth）
    if (!nextContext.currentUser) {
      // 带用户前往登录页
      RouterActions.unauthedUserStart();
    } else if (nextContext.currentUser) {
      // 带用户前往他们的培训主页
      RouterActions.authedUserStart();
    }
  }
  ...
 }
```

这个方法的好处在于不需要使用任何的回调函数或者promise。`Launch`组件内的全部代码都是同步的，只需要监听`AuthState.Store`的变化并确保能调用相应的`componentWillUpdate`方法即可。

## 3.5 iOS 平台的环境配置

通常情况下，需要根据不同的构建过程来快速修改应用的配置变量。比如你想要先构建与staging环境服务器通信的应用，再构建与生产环境服务器通信的应用。遗憾的是，使用iOS平台的React Native时，没有内置的途径能够方便地切换配置。

3

为了解决Myagi遇到的这个问题，我们结合使用了几种技术：iOS构建配置与scheme文件、自定义构建脚本、environment.plist文件以及`react-native-env`库。为了展示如何使用这些技术，接下来将以切换API服务端URL的过程为例，讲解如何把服务端从开发环境切换到staging环境，再到生产环境。

### 3.5.1 plist 文件与 `react-native-env` 模块

plist（属性列表）文件通常用来保存应用的配置信息。实际上，所有全新的iOS工程都带有默认的Info.plist文件，里面包含了应用相关的详细配置（例如Bundle Verison）。这些文件可以在应用内很方便地通过Objective-C（或Swift）来访问。举例来说，假如想要从Info.plist文件中获取Bundle Version值，可以运行以下的Objective-C代码。

```
NSDictionary *plistData = [NSDictionary dictionaryWithContentsOfFile:[[NSBundle
mainBundle] pathForResource:@"Info" ofType:@"plist"]];

/*打印保存在Info.plist文件中的'Bundle Verison'值*/
NSLog([env objectForKey: @'Bundle Version']);
```

plist文件的值也可以在React Native使用的JavaScript执行环境中访问。我们可以使用`react-native-env`模块来实现。`react-native-env`模块允许创建名为environment.plist的文件，该

文件中的值可以被React Native的JavaScript执行环境访问。举例来说，假如environment.plist文件中有一个APIServerURL属性，可以用以下代码在控制台输出该值。

```
import env from 'react-native-env';

env.get('APIServerURL').then( val => console.log(val) );
```

因为plist文件里的值可以被JavaScript执行环境和Objective-C/Swift代码访问到，所以可以合理地用它们来放置（至少一部分）配置信息。我们在Myagi中的做法正是如此，尤其是针对动态变量而言，我们想要根据开发应用的计算机来设置它们（比如想要在开发过程中让APIServerURL包含当前计算机的IP地址）。不过，在构建过程之间修改environment.plist文件里的变量，需要用到scheme文件、构建配置以及自定义构建脚本。

## 3.5.2　iOS scheme 文件与构建配置

scheme构建文件仅仅是一个“已保存”的设置集合，用于在Xcode中构建和运行应用。你可以在Xcode窗口左上方的下拉菜单中快速改变当前的scheme。




如上图所示，可以点击左侧带有Myagi标签图片的按钮在不同的scheme文件之间切换。你也可以在这里创建新的scheme文件或复制已有的。

为了实现在不同构建过程之间修改配置变量，scheme文件就显得很有用了，因为它们允许我们快速切换不同的构建配置。构建配置是编译iOS应用时用到的设置集合。所有全新iOS工程的默认构建配置为Release和Debug两项。在工程设置的Configurations菜单中，可以很方便地添加新的构建配置，或复制已有的。



在Myagi工程中，我们使用不同的构建配置来修改配置变量。如上图所示，我们有`Debug-StagingAPI`和`Debug-ProdAPI`两项配置，根据所选的配置，应用就会使用不同的API服务端URL。要创建新的配置，只要复制一个已有的配置（`Debug`或`Release`），进行恰当的命名，然后创建一个使用该配置的scheme文件（或是复制一个已有的）。要改变与scheme文件关联的构建配置，只要编辑该scheme，找到`Build Configuration`菜单，选择相应的配置即可，如下图所示。

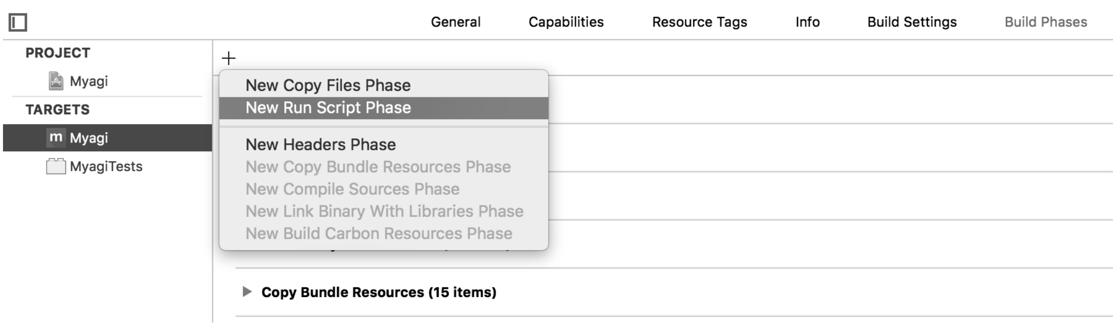



有了新组合的构建配置与scheme文件，接着就可以用自定义的构建脚本为该scheme文件设置正确的配置变量。

### 3.5.3　自定义构建脚本

在Xcode中可以添加自定义的shell脚本，作为应用构建过程的一部分来运行。具体的操作步骤是，在文件树中点击你的工程 > 选择工程的target > 点击build phases > 点击+按钮 >点击New Run Script Phase，如下图所示。






这个脚本将会在构建过程中运行，可以用它在environment.plist文件中写入值，以便通过`react-native-env`模块来访问。

为了知道写入什么值，构建脚本需要知道当前的构建配置，幸运的是可以通过`CONFIGURATION`环境变量来取得。因此，构建配置的值决定了配置变量的值，并且可以用scheme文件来快速切换构建配置。以下是Myagi构建脚本的精简版，只包含了设置`APIServerURL`值的代码。

```
set -e

# 设置environment.plist文件的路径
env_plist=$dir/environment.plist
```

```
# 设置api_server_url的默认值
api_server_url="http://localhost:8000"

# 通过CONFIGURATION环境变量决定当前的构建配置，然后根据配置决定api_server_url的值
if [ "$CONFIGURATION" == "Debug" ]; then

  # 如果是debug配置，使用当前计算机的本地IP地址作为服务器地址
  api_server_url="http://`ipconfig getifaddr en0`:8000"

fi
if [ "$CONFIGURATION" == "Release" ]; then

  # 如果是release配置，使用生产环境的API
  api_server_url="http://myagi.com/api/v1"

fi

# 使用PlistBuddy程序把API服务端的URL值写入environment.plist文件，
# 以便在JavaScript执行环境内访问它
/usr/libexec/PlistBuddy -c "Set :"APIServerURL" ${api_server_url}" $env_plist
```

这个脚本完成后，就能保证为当前配置正确设置了API服务端URL的值，并且能通过切换scheme文件来方便地修改配置。最后，在应用内就可以使用`react-native-env`模块来获取并使用URL值。

## 3.6　跨平台代码共享

React Native的主要优势之一在于，它使得iOS应用、Android应用以及Web应用的代码与资源可以共用。由于框架用JavaScript编写，既可以在Web浏览器也可以在原生应用中执行。实际上，就像本章开头所说的那样，React Native最吸引Myagi团队的方面就是跨平台代码共享的能力。

当然，代码共享并不仅仅是把一个平台的代码复制到另一个平台上，然后点击一下“运行”那么简单。实际上，不同平台之间不能共享整个完整的代码库，只有其中的一大部分可以。对每个平台来说，总会有一部分代码只适用于它自己。

这绝对可以算一个优点，鉴于用户对iOS、Android以及Web应用在视觉与使用体验上有不同的期待。举例来说，我们希望iOS应用可以遵循iOS设计指南（https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/）,而Android应用要遵循Android设计指南（https://developer.android.com/design/index.html）。遵循这些不同的设计指南，就必然意味着你的应用在不同平台上有不同的视觉与使用体验。

或者，你可以选择无视这些设计指南，开发一个在iOS、Android以及Web浏览器中看起来一样的应用。这种做法绝对可行；像Phonegap和Trigger.io这样的技术就可以用兼容Web的语言（HTML、CSS与JavaScript）来开发应用，并在不修改代码的情况下部署成原生的iOS和Android应用。然而，因为前面提到的对不同平台的期望存在差异，这种“一次性开发，全平台运行”的

方式会带来一些问题。此外，在某个平台上完美运行的代码很可能在另一个平台上运行得很糟糕（尤其CSS方面经常出现这种问题）。试图一次性开发应用并让它可以在全平台上运行，必然意味着要处理不断出现的设备特定问题和变化。

React Native（以及React）的设计理念不是让开发者"一次性开发，全平台运行"，而是"一次性学习，全平台开发"，这种理念在于你学习了React和React Native之后，可以用这些技能为任何支持的平台开发应用。事实上，如本章前面提到的，React Native确实支持代码共享，并且也不只是"一次性学习，全平台开发"，它还支持"一次性开发大部分代码，全平台运行"的方式。正因为如此，React Native使得开发团队能够使用自身所有的技术栈，还能跨平台共用部分代码，同时能够方便地开发用户界面，让视觉与使用体验都符合运行环境的期望。

## 3.6.1　代码共享的利与弊

在深入了解那些跨平台代码共享的技巧之前，值得先提一提这种做法的利与弊。

正如之前提到的，跨平台代码共享的益处非常明显：你不需要为不同平台一遍又一遍地重写那些本质功能一样的代码。而且，你只需要学习一套共享模块，不用学习如何在各个平台上完成同样的事情。

然而应用之间共享大量代码也存在弊端：不断增加的依赖项。一段代码复用的程度越高，对它的依赖也就越多。如果这段代码有一些改动，那就要考虑改动会对用到这段代码的所有地方造成什么样的影响，这非常重要。当然，应用中涉及代码复用的场景都需要处理这样的问题，然而代码在多个平台与环境共享时，随着可能的依赖项逐渐增加，这个问题就没那么简单了。一个改动可能在某个平台上有效，但在另一个平台上可能会出现问题，你在修改代码时要牢记这一点。



这种依赖问题的解决方案有以下两个。

- 全面的测试流程→自动化测试做到位，你就能轻松得知共享代码里的改动不会对特定平台的某个特性造成影响。
- 对多个平台之间共享的代码进行修改的流程要清晰。

接下来介绍一下跨平台共享代码的流程，随后的章节中会介绍测试。

## 3.6.2　iOS 与 Android 间的代码共享

开始学习原生应用和Web应用之间的代码共享之前，先来介绍一下iOS和Android之间的做法。React Native共享iOS和Android的代码有两种方式能做到：使用特定平台的扩展，以及内置的`Platform`模块。

### 1. 特定平台扩展

React Native提供了特定平台扩展这一有用的特性。比方说有一个`Button`组件，我们希望它在

iOS和Android平台上的视觉以及使用体验完全不同，实际上可以为两个平台定义两个独立的文件：button.ios.js与button.android.js。接着就可以在每个平台各自对应的文件中定义`Button`组件，然后由React Native负责按照平台加载正确的那个。其他用到该`Button`组件的文件或组件不需要任何改动：它们只要像`Button`组件放在button.js文件中那样调用它即可，React Native会确保它们在当前平台上可以取到正确的`Button`组件。

#### 2. `Platform`模块

有时候我们并不想为每个平台完全重新定义一个组件。比方说你只想对一个组件做一点小小的修改，使得它在Android和iOS上的行为或者呈现方式上略有不同，这个案例中为每个平台单独创建一个文件将很繁琐。这就轮到原生的`Platform`模块派上用场了。假设我们想要按照平台稍微地调整组件中的文本，可以像下面这样做。

```
import { Platform, Text } from 'react-native';

...

function WelcomeText() {

 const text = Platform.select({
  ios: 'Welcome to our iOS app!',
  android: 'Welcome to our Android app!'
 });

 return <Text>{text}</Text>
}
```

如代码所示，`Platform.select`方法将为当前的平台选择正确的文本值。

### 3.6.3　原生应用与 Web 应用间的代码共享

共享原生应用和Web应用的代码库会更有挑战性。这种做法的目的在于减少两个代码库之间的代码冗余，同时还要保证共享代码的改动不会对依赖的代码库造成严重问题。

第一步，先要判断什么代码可以共享，然后制定共享这些代码的最佳计划。理论上其实可以共享所有满足下列条件的代码。

- 没有利用或引用那些指定环境特有的组件。例如，引用了React Native内置`View`组件与`Text`组件的代码，它们不可能与运行React的Web应用共享。
- 没有引用环境特有的库或内置工具。这种情况主要指浏览器中`document`这样的全局变量，它们在React Native的JavaScript执行环境中不存在。另外，部分的内置工具，比如`fetch`方法，可以通过腻子脚本（polyfill）实现，因此用到这类内置工具的代码就可以跨环境共享。

除了这些限制以外的所有代码，至少在理论上都可以共享。但实践中把满足共享条件的代码都进行共享也不太合理，不过在Myagi应用中，我们在React Native应用和Web应用之间设法共享了代码库的以下部分。

- 所有状态管理代码。也就是用于获取和更新数据的所有模块。
- 所有样式变量，比如颜色信息。
- 所有工具模块。比如`validators`模块，它包含了检测字符串是否为有效的邮件地址或URL的方法。
- 所有国际化（i18n）模块。它的方法用于获取某个字符在特定语言下的翻译，组合所有需要的翻译文件来实现该需求。
- 一些组件内的方法。比如`Form`组件与相关的`Input`组件包含了大量的表单验证以及提交逻辑。与其在不同平台上复制这些逻辑，不如像我们所做的那样，把所有可共享的逻辑抽取出来，放到`FormMixin`和`InputMixin`中，以便供原生组件或Web组件使用。

因为我们开发Web应用比React Native应用早了一年多，共享这些代码使得React Native应用的起步更加简单。Web应用中诸多问题的解决方案，由此可以用在原生应用里。实际上，利用这些代码减少了第一版应用从开始开发到最终发布的时间，只用了一名工程师三周左右的时间。这对我们是一个极大的鼓舞，并且将来任何时候我们想要为原生应用或Web应用开发一个新的特性时，还能从代码共享中受益。

3

#### 使用Git共享React Native与Web应用之间的代码

共享原生应用与Web应用的代码有很多方式。以一个很简单（但存在问题）的方案为例，只要把一个代码库的代码复制到另一个里就行。一开始这么做倒是可行，然而一旦在其中一个代码库做了改动，想要另一个也包含这些改动，那就不得不手动复制粘贴，这很容易出错。显然，这不是一个好方案。

另一个方案是把Web应用和原生应用的代码库放在同一个仓库中，然后把所有共享的代码放到仓库中的一个共享文件夹下。这样两个代码库都可以引用这个共享文件夹，对共享代码的小改动也就能更新到两个代码库。这个方案更好，然而它还是不够完美，正是因为前面提到的依赖问题：需要检查对共享代码做的任何修改，来确保不会引发其他代码库的问题。这绝对能够做到，但是我们还是决定不用这个方案，因为把错误代码放进生产环境中的风险实在太大了。

我们最终选择的方案是使用Git来管理Web应用和原生应用这两个独立仓库之间的共享代码。Git被设计成可以很方便地共享多个地方的代码，因此这个方案很适合我们。方案内容便是在主Git仓库内使用子仓库。举例来说，当需要共享`utilities`模块时，采用以下操作步骤。

- 打开`webapp`模块下的`utilities`模块。
- 在此处初始化一个空的Git仓库，提交这个新仓库并将其推送到Git服务端（我们使用bitbucket.org）。
- 打开原生应用代码库中我们想要复制`utilities`模块的文件夹。

- 用`git clone`命令从远端拉取模块。
- 重要的是，使用以下命令，把新的代码添加到原生应用的主仓库中：`git add utilities/ && git commit -m "Added the utilities module"`。

最后一条命令中最重要的是`utilities`后面的正斜杠。如果遗漏了正斜杠，那么Git会把utilities文件夹添加成`git`子模块。这样仍然可以达到我们的目的，然而我们在实践中发现Git子模块用起来很繁琐。相反，加上了正斜杠后，Git会把`utilities`模块当作普通文件夹，并忽略它实际上是另一个Git仓库的事实。

现在我们要对原生应用代码库内的`utilities`模块进行修改，接着把这个改动同步到Web应用的代码库中，步骤如下所示。

- 打开原生应用代码库内的utilities文件夹。
- 提交并推送任意的改动。
- 打开Web应用代码库内的utilities文件夹。
- 用`git`拉取更新。
- 如果Web应用代码库的`utilities`模块已经做了一些修改，在合并时可能存在冲突。如果这样，请解决冲突并合并。
- 如有必要，运行测试来确保这个改动没有破坏Web应用的某些功能。
- 提交更新了`utilities`模块的Web应用代码。

这个方法的好处在于它相当简单，并意味着你可以修改共享代码库，还不用担心它是否会对其他用到这段代码的地方造成影响。你要做的仅仅是同步改动的时候对一切进行测试。

## 3.7　测试

React Native的强大特性之一就是允许快速迭代应用。这主要是缘于一些可以让React Native开发者受益的优势，并且传统的iOS以及Android开发者无法享受到这些优势。它还有一项能力，就是每次修改后可以轻松刷新应用界面（最近还可以热重载应用，连刷新都不需要了）。这使得开发过程中可以很快地修改并优化应用。此外，通过使用微软的CodePush平台，实际上开发者几乎能够立刻把改动推送给用户，无需等待任何应用商店的审批过程。

如此快的开发周期需要警示一点：制定恰当的测试过程尤为重要。当你的团队成员通过一条命令就能更新所有用户的应用时，你需要确保成立完善的自动化（也可能是手动的）流程以保证更新不会影响功能。这就必然需要一个健全的测试过程。幸运的是，有一些不同的技术可以让React Native的测试相对没那么痛苦。

## 3.7.1　测试类型

软件开发过程有几种不同的测试类型：单元测试、集成测试、金丝雀测试（canary testing）[①]以及质量保证测试等。为React Native应用制定测试流程计划之前，重要的是认识到每种测试类型的利与弊，这样才能决定如何最好地利用每种测试以及在哪里下最多的功夫。

在Myagi应用中，我们把大部分心血都投到了集成测试上，接着是单元测试，最后是质量保证测试。下面将概述每种测试类型。

### 1. 单元测试

传统上来讲，单元测试就是对单一功能进行测试。它们针对应用的最小可测试部分，也可以看成组成逻辑的单元。实际中，单元测试通常意味着对单一的函数或者方法进行测试。举个简单的例子，下面的基础JavaScript代码展示了一个函数以及与它相关的单元测试。

```
function getAverageAge(users) {
 /*
  如果users是带有age属性的对象的数组，该函数将返回平均年龄
 */
 let total = 0;
 users.forEach(user => total += user.age);
 return total / users.length;
}

function testGetAverageAge() {
 /*
  该函数执行后会测试getAverageAge函数，如果它没有返回期望值，将抛出错误
 */
 let result = getAverageAge([{
  age: 10
 }, {
  age: 20
 }];

 if (result !== 15) {
  raise new Error('Test failed');
 }
}
```

在这个普通的示例中，`getAverageAge`函数是单一的功能单元，由单元测试`testGetAverageAge`进行测试。

然而考虑一下更复杂的情况，即被测试的功能单元调用执行了其他单元。比如，更新`getAverageAge`函数的最后一行代码。

① 这种测试只把改动的代码推送给一小部分用户，并且用户不知道自己参加了此项测试。该测试目的在于让改动的代码可以跑在真实环境中，效果更直观。详情请参见网页https://www.quora.com/What-is-Canary-testing。
——译者注

```
...
// safeDivide函数在第二参数不为0的情况下，用第一参数除以第二参数。
// 如果第二参数为0，就返回0，而不是JavaScript除法默认的Infinity
return safeDivide(total, users.length);
...
```

如果还用原来的testGetAverageAge函数来测试更新后的getAverageAge函数，就不只是测试一个功能单元了，而是至少测试了两个：getAverageAge函数，以及新的safeDivide函数。另外，safeDivide函数可能还调用了其他函数来实现自身的功能，也就是说也要对它们进行测试。

stub（打桩）是这种问题的标准解决方案。stub其实就是在测试过程中暂时“重写”其他函数，从而保证它们可以返回特定值，也就解除了被测试函数对其他功能单元的依赖。下面更新最初的单元测试代码来使用一下stub。

```
function testGetAverageAge() {
 // 保存最初的safeDivide函数引用，在测试的最后还原它
 const origSafeDivide = safeDivide;
 // stub safeDivide函数，这样它内部的简单逻辑就由我们掌握
 safeDivide = function(a, b) {
  return a / b;
 }
 // 运行最初的测试
 let result = getAverageAge([{
  age: 10
 }, {
  age: 20
 }];

 if (result !== 15) {
  raise new Error('Test failed');
 }

 // 还原最初的safeDivide函数
 safeDivide = origSafeDivide;
}
```

现在testGetAverageAge函数只会测试一个功能单元，由于对safeDivide函数进行了stub，当对它进行单元测试时，其实已经没有什么内部逻辑了。

上面解释了单元测试的概念，不过还没有介绍它们的优势。大体上说，单元测试有以下几种优势。首先，它们促使你按照单一职责原则来编写函数、方法和类，这使你的代码更容易理解与修改。其次，它们让代码中独立函数或方法的重构变得更简单，因为这样一来你就能肯定重构不会改变该函数的行为。最后，它们其实也能作为代码的文档，因为它们显式地定义了在传入某些参数时函数和方法的预期行为。

然而单元测试也有缺点，它们无法确保你的应用在用户使用过程中一定能正常工作。所有测试过的功能单元都正常工作，并不意味着它们联合起来时整个应用可以正常工作。而这时就该集成测试发挥优势了。

### 2. 集成测试

单元测试侧重于独立功能单元的正确性，而集成测试侧重于测试多个功能组合到一起的运行情况。实际上，两种测试绝非截然不同。它们其实就是介于单一功能测试（单元测试）和大量功能组合测试（集成测试）之间的一系列测试中的两种极端情况。也就是说，两者之间的区别很重要，因此在编写测试时务必要针对两种极端情况之一来进行。那些介于完整集成测试与单元测试之间的不伦不类的测试，往往两种测试的优势都得不到。

那么集成测试的优势是什么呢？一般来讲，其优势在于保证了提供给用户（可以是传统意义上的用户，也可以是另一个使用你编写的模块的软件开发者）的产品功能一致且可用。它们试图确保你所编写并且经过单元测试的所有不同功能单元能一起正常工作。以下代码展示了一个电子商务网站用户界面的集成测试，用到了一系列虚构的辅助函数。

```
function testUserCanPurchaseItem() {

 visitURL('/product/123');
 clickOnElement('#addToCartBtn');
 visitURL('/myCart/');
 clickOnElement('#buyBtn');

 // 以编程方式输入购物详情
 ...

 clickOnElement('#finalizePurchaseBtn');
 // 如果没有找到purchaseSuccessTxt元素，该函数将会抛出异常
 assertElementExists('#purchaseSuccessTxt');

}
```

以上代码使用了虚构的库，我们可以通过它来编程模拟用户操作，在应用界面内导航。接着测试用户是否能做或者看到某些东西。这和之前介绍的单元测试差别很大，因为渲染用户界面每个部分或者处理每次交互所需要的功能单元的数量可能非常多（10多个甚至100多个函数、方法以及类）。集成测试的目的是忽略有多少不同的功能单元要进行测试，从而保证从用户的角度上看，应用能够正常工作。

### 3. 质量保证测试（QA）

我们在Myagi中用到的最后一种测试类型便是质量保证测试。它和集成测试很像，只不过由人来手动进行。这种测试类型的目的与集成测试相似，就是要从用户的角度测试应用的所有功能可以正常工作。区别在于，质量保证测试需要真正的人来进行，因为人可以发现那些设计上无法被集成测试发现的问题，这也是这种测试的优势。回到之前电子商务网站的例子，页面上的购买按钮可能被意外地移动了位置，因此尽管它在视觉上偏移了，但还是可以点击。集成测试中除非明确指明要测试这一点，否则无法发现这个问题，而质量保证测试就可以做到。当然质量保证测试也有缺点，它需要人力以及时间，而集成测试与单元测试都是自动化的。因此，进行质量保证测试的频率没有其他测试类型那么高。

### 3.7.2　单元测试的实现

在Myagi应用中，通常选用单元测试来测试那些整个应用都普遍使用的函数与方法，并重点关注那些Web应用和原生应用之间共享的部分。我们想要确保这些共享的功能单元始终正确，并确保修改它们不会在其他代码库中引发意外问题。同时，单元测试也促使我们把共享代码设计成只有单一职责的独立函数与类，这样使理解与跨平台使用它们都变得更简单。

我们选择了测试框架Mocha（https://mochajs.org/）搭配断言库Chai.js（http://chaijs.com/）来帮助编写单元测试。Mocha包含了一些定义测试的基本函数，还附带了命令行程序方便执行测试。使用Mocha编写的基本测试集合如下。

```
describe('MathModule', function() {

 it('can add numbers', function() {
  ...
 });

});
```

如代码所示，describe函数内定义的每个it函数可以看成独立的测试，整个集合可以当作为特定模块或功能集合编写的测试套件。只要定义了这些测试，并放在test目录下，就可以执行Mocha命令行工具来运行它们。

断言库Chai可以与Mocha一起使用（也有很多其他的断言库可以用）。它内置的函数可以用来检查测试过程中的值是否正确。我们用Chai来重写前面的例子，如下所示。

```
describe('MathModule', function() {

 it('can add numbers', function() {
  const result = MathModule.add(1, 3);
  expect(result).to.equal(4);
 });

});
```

如代码所示，Chai提供了expect函数，我们用它来测试add方法返回的值是否正确。

最后用到的一个重要的测试工具是Sinon（http://sinonjs.org/）。这个库可以很方便地stub函数，模拟对象甚至模拟服务端返回的请求。以下代码来自示例应用中的一个测试，它展示了如何组合使用这些工具。所有用stateDefaultGenerator函数生成的store，都会添加一个getItem方法，测试的对象就是它。注意，TestUtils.server方法在Sinon的fakeServer模块上做了一层简单的封装。

```
...

beforeEach(function(){
  // 用sinon.fakeServer初始化服务端
  TestUtils.server.create();
});
```

```
...

it('can get an item by id', function(done){

   // 调用异步的getItem方法
   FoosState.Store.getItem(1).toPromise().then(function(item){

     // 确保取回的数据项有正确的name属性
     expect(item.get('name')).to.equal('foobar');

     // 通知Mocha测试结束
     done();

   });

   // 调用getItem时，使用sinon触发响应
   TestUtils.server.respondWith(/foos\/1\//, {
     id: 1,
     name: 'foobar'
   });

 });
```

在以上代码中，我们测试了用于获取Foo数据项的`getItem`方法。由于`FoosState.Store`需要调用API来获取数据，我们用`TestUtils.server`对响应进行了stub。这样一来，可以保证执行测试时无需依赖API服务端的运行。



### 3.7.3 UI 集成测试的实现

虽然我们的应用只需要Mocha和Chai就能搭建单元测试，但集成测试显然更加复杂。这是因为集成测试需要编译应用，然后通过编程与UI进行交互，接着测试它是否如期望的那样正常运行。此外，还要保证应用有数据可以展示，这意味着需要搭建模拟API服务器，还要保证服务器能为每次测试返回一致的数据。

幸运的是，taskrabbit/ReactNativeSampleApp（https://github.com/taskrabbit/ReactNativeSampleApp）这个完美的代码仓库提供了一个解决方案，对上面提到的一些问题做了处理，我们几乎完全根据这个方案搭建了集成测试流程。

我们在集成测试中用的关键工具是Appium。Appium是一个UI自动化工具，它可以有效地让应用在iOS模拟器中运行，并通过JavaScript“控制”与应用的交互。这个过程还结合了一个用Koa创建的简单模拟API服务器。有了模拟API服务器，我们就能在应用尝试访问API时准确地定义要响应的数据。

以下登录界面的测试展示了如何组合使用这些工具。

```
it('should allow login with valid data', function* (driver, done) {
```

```
    // 获取页面上的元素
    let email = yield driver.elementById('Email');
    let password = yield driver.elementById('Password');
    let submit = yield driver.elementById('Submit');

    // 更新输入框内容
    yield email.setImmediateValue('test@test.com');
    yield password.setImmediateValue('123456');

    // 当应用尝试登录时，通知服务端返回成功的响应
    server.post('token_auth', { token: '123' });
    server.get('users/current', fixtures.currentUser());
    server.get('training_plans', []);

    // 触发登录操作
    yield submit.click();

    // 过渡到有访问权限限制的页面
    yield driver.elementById('Training');

    // 通知Mocha测试结束
    done();

  });
```

尽管这段测试代码很短，它还是做了很多事情。首先，需要重点关注的是定义该测试的函数是一个生成器。从function关键字（function*）后面带有的星号可以得知。生成器（generator）是JavaScript中的新特性，它让函数的调用者可以有效地“暂停”函数的执行，并在之后某个时刻恢复。这种情况下使用生成器让我们的测试定义非常整洁，除此以外可能没有其他的方式能做到。这个测试过程需要暂停的理由在于通过Appium访问元素是异步的。举例来说，尝试用driver.elementById('Email');这行代码访问邮件地址输入框，这实际上就是异步执行的。正因如此，需要在获取元素时“暂停”测试过程，等取回元素后再继续执行下一行。生成器允许只用yield声明就能实现这点。每条yield声明表示“执行当前的异步声明，等待完成后再接着执行后面的”。

使用yield声明，我们可以检查屏幕上是否存在特定的元素。这就是上述代码的前三行实现的内容：找到带有Email、Password还有Submit标签的输入框。接着，再结合driver对象和yield声明来设置邮件与密码输入框的值。然而在通知Appium按下Submit按钮前，我们实际上用server模块定义了在特定的API路径被访问时服务端要返回的内容。举例来说，我们知道应用尝试登录时，会向token_auth路径发起一条请求，并且如果登录请求成功了，应用希望返回一个令牌。因此，指示模拟服务器（由Koa搭建）完全这样来做。

一旦指示服务器要返回正确的值，就能调用yield submit.click()，这行代码通知Appium点击提交按钮。接着应用会调用由Koa模拟的API路径，再接着就会返回表示登录成功的值。最终，检查一下应用是否过渡到了Training界面，这正是登录成功后的期望行为。

于是，你见到了如何组合使用所有不同的技术要素（Appium、Mocha、Koa以及Chai等）来定义清晰明了的集成测试。另外，尽管集成测试很有效而且非常容易编写，它们要花大量时间去运行。实际上，仅仅编译应用并启动模拟器就要花超过一分钟的时间，而这正是执行单个测试之前的必要过程。因此，集成测试的执行可能很慢，并且这样做需要面对缓慢的迭代周期。

### 3.7.4 QA 测试

QA测试是应用的改动部署给用户之前的最后一道防线。正如之前提到的，这个过程完全手动进行，但我却要极力推荐。保持QA测试一致性的简单方式，就是在电子表格或者共享文档中定义一系列测试人员必须要操作的“测试用例”。在将改动部署给用户之前，开发者必须简单地操作这些手动测试，并确保它们能够“通过”（即没有任何差错）。

尽管QA测试确实稍微减缓了部署过程，它仍然是一种很好的方式，让你对应用的所有核心功能可以正常运行有充分的信心。而且，它通常能够暴露出应用在UI和UX方面的问题，你可以选择在发布之前或者在下一个版本中进行修复。这只是确保应用质量能不断提升的一种方式，你要像你的用户那样来体验自己的应用。

## 3.8 发布与更新



传统上来看，更新原生应用的过程很慢，尤其是更新iOS应用的情况。iOS平台主要是应用商店审核流程的缘故。每个提交给商店的应用和更新版本必须经过人工审核，这就意味着有时候提交应用后要经过很多天，它才能被用户获取。甚至发布一个更新版本，也要经过一段时间，所有用户才能都下载到新版本，特别是用户没有启用自动更新时。

幸运的是，有了React Native，再利用微软的CodePush服务，可以显著地减少更新耗时。接下来将大致介绍一下应用发布过程中如何结合使用微软CodePush、传统应用商店更新流程以及Git工作流。

### 3.8.1 Git 工作流

Git工作流（http://nvie.com/posts/a-successful-git-branching-model/）其实就是使用Git的一种方法论。Git本身的基础知识超出了本书的范畴，不过下面会简单介绍一下Git工作流。

- 有两条主分支，`develop`与`master`。`master`分支是发布给用户的应用最新版本。`develop`分支用于日常的开发。
- 每当要开发重大的新特性时，基于`develop`分支创建特性分支。完成开发后，将特性分支合并到`develop`分支。
- 每当要为用户发布更新时，将`develop`分支合并到`master`分支，通常也会使用`release`分支作为中介，接着部署更新。

- 如果因为发现了bug或者其他问题，需要更新应用，只需简单地基于`master`分支创建`hotfix`分支，完成修复，将更新后的代码合并到`master`与`develop`分支，然后基于`master`分支为用户推送更新。

按这些规则使用Git的好处在于，它们促使我们在`master`分支上始终保存着应用最新最活跃的版本，而且可以很方便地获取。这意味着能够快速地创建`hotfix`分支来修复bug，无需把最近更新的代码版本也进行部署。这一点很重要，因为原生应用的缓慢更新流程通常意味着，用户手机上运行的应用版本与开发者正在开发的最新版本之间有很大差别。Git工作流提供了很多方便，可以快速"切换回"应用的活跃版本修复bug，接着再回头开发应用的最新版本。接下来对Git工作流进行更全面的描述。

### 3.8.2　iOS 应用商店更新流程

尽管Apple提供了应用商店更新流程的详细文档，还是值得在Git工作流的背景下重新理解一番。Myagi团队中这个过程大致如下所示。

- 约定一个想要在应用商店发布更新的日期。
- 更新日期到来时，基于`develop`分支最新的提交版本创建`release`分支。
- 在`release`分支上，确保单元测试与集成测试都能成功运行。
- 手动执行QA测试用例，并根据新增特性增加必要的新用例。
- 修复上一个过程中出现的任何问题，接着重复测试一遍。
- 至少两名开发者审核了应用目前的状态后，才能将其提交给应用商店。
- 这一步，`release`分支的使命结束，也就是：提交最新的改动，把它们合并到`master`与`develop`分支，然后删除`release`分支。
- 接着，通过Xcode编译应用，并提交给应用商店等待审核。
- 最后，返回`develop`分支，为应用创建一个版本号。这样可以确保我们下次进行更新时，有对应的新版本号。

这个过程很好地确保了应用的绝大多数问题能在暴露给用户之前得到修复。确保这一点非常重要，因为如果发布后出现了问题，很可能要重新进行完整的审核才能把修复推送给用户（虽然Apple每年都允许你做几次紧急修改）。

不过偶然情况下bug也会出现在发布的版本中。幸运的是，这种情况下可以使用微软的CodePush，独立于应用商店对应用进行更新，因此就能绕过应用商店审核流程所必需的等待时间。

### 3.8.3　CodePush 更新流程

比起用Swift开发iOS应用以及用Java开发Android应用，React Native的主要优势之一便是可以使用微软CodePush（https://github.com/Microsoft/code-push）。有了CodePush，你就能远程托管

JavaScript打包文件和资源文件（放在微软提供的服务器上），因此当用户打开应用时，他们的设备就会检查JavaScript文件（或者图片等资源文件）的改动，然后自动更新。这是一项很强大的特性，因为它意味着你的改动几乎能立刻被用户获取，无需等待Apple的批准或者Google的分发。而且，Apple与Google实际上允许这种应用更新方式，也就是说这样做不会导致你的应用被iOS或Android应用商店禁止或拒绝。

以下是Myagi的操作流程。

- 当准备用CodePush发布新版本时，还是像平常一样基于`develop`分支创建`release`分支。然而，由于可以用CodePush进行快速更新，实际上发布版本会更加频繁（大概每个新特性都会发布一次），而不用在发布前等待一堆特性完成。
- 像往常一样运行测试。不过随着发布的改动减少，出错的概率会更小，这意味着测试流程的耗时要比通过应用商店进行更新少了很多。
- 准备就绪后，`release`分支完成使命，接着通过一系列命令把应用部署给用户。
    - 进行编译的代码如下所示。

    ```
    node --max-old-space-size=4096 node_modules/react-native/local-cli/cli.js
    bundle --platform ios --entry-file index.ios.js --bundle-output ios/release/
    main.jsbundle --assets-dest ios/release/ --dev false
    ```

    - 接着通过CodePush进行发布的代码如下所示。

    ```
    code-push release -d Production myagi-ios ios/release
    ```

    - 当然，实际上我们不需要记住这些命令并每次运行，它们被组合成一条更简单的命令，添加到package.json文件的`scripts`部分，这样实际运行的部署改动命令就简单多了，如下所示。

    ```
    npm run code-push-deploy-prod
    ```

- 最后只需快速检查一下，从应用商店下载应用以确保部署成功。要让推送给CodePush的更新被用户“安装”有多种不同的方式。结合本例，只要用户打开了应用，应用就会检查更新。等他们下次打开应用时，更新就会安装完成。这样不在应用打开时立刻进行更新，可以减少对用户产生的影响，但又完全保证大多数用户能得到更新（因为更新可以自动安装，用户无需面对提示框）。

这种流程允许在开发过程中实现持续交付。持续交付是指更新能以可持续的方式快速安全地发布。也就是说发布给用户的可以是少量更新，而不是大量改动。它意味着我们的软件能不断为用户优化，同时也减少了每次更新带来的风险：更少的更新意味着改动给用户造成主要问题的可能性更小。

还要提一点，使用微软的CodePush并不是说不要通过应用商店更新应用。我们仍然必须定期进行常规的iOS更新流程，理由有以下两点。

3

(1) 下载应用的新用户应该立刻拥有一个近期的版本。我们不希望他们下载应用后，还要等待最新的CodePush更新，然后再重启应用。

(2) CodePush只允许我们更新JavaScript文件以及其他资源文件，不允许重新编译应用的Swift/Objective-C部分。这意味着如果我们为应用添加了新的原生组件或模块（通过第三方库或自己开发），那么只能通过App Store的发布流程来部署这种改动。

### 3.8.4　小结

以本书介绍的方式使用Git工作流、微软CodePush以及iOS应用商店的更新流程，让我们能够在向用户发布更新之前很大程度上精简团队数量。同时，在测试流程的帮助下，我们能有充分的信心，发布给用户的任何改动都不会严重损害他们的体验。

第 4 章

# 示例应用：TinyRobot

几乎每个人都使用消息应用，并且大部分用的是移动端版本。但是很少有消息应用把消息功能和位置功能集成到一个产品中。附近有谁？你的朋友在哪里？他们在做什么？

TinyRobot是iOS平台上基于位置的消息应用。它仍处于开发阶段，不过大体上已经完成了。

## 4.1 为何选择 React Native

除了Apple Xcode内置的Objective-C和Swift开发方式，如今市面上还出现了大量的开发平台（React Native、PhoneGap、Appcelerator Titanium、Ionic框架、Intel XDK、NativeScript 、DevExtreme、Reapp、Xamarin等），真的很难选择。最简单的方式就是用Xcode来开发，然而原生开发的速度非常不理想。

最困难的部分莫过于根据UI设计师的原型来调整原生代码。Xcode内置的UI Storyboard编辑器可以派上用场，但UI设计师不会使用它，因此这项任务就落到了开发者的肩上：调整间距、字体、颜色、UI、部件以及布局。也有一些现成的代码生成器可供使用，不过它们能做的与我们的期望相差甚远。此外，开发者在每次调整后都要重新编译应用，这就更慢了。

原生开发还有另一个缺点：不能进行真正的组件化开发。分布式团队用iOS Storyboard进行开发会相当困难，另外也很难把应用拆分成独立的组件来分别开发。CocoaPods产品尝试解决这个问题，但还是明显影响了开发速度。你可以把代码放到Cocoapod组件当中，在修改每个组件后重新构建应用，但这样做可能不时会发生各种各样的构建问题。



第三个缺点便是漫长的部署过程，而且即使修复很小的bug也必须经过AppStore的审核。

另一方面，Web开发就没有这些问题，而且很多产品都尝试把Web开发方式带到iOS/Android应用中。PhoneGap带头做了这种尝试，它允许开发者开发HTML页面，然后嵌入到iOS的WebView容器中。不过，所有基于WebView的产品都有一个明显的缺点，即性能远不如原生。应用的Web控件与动画看起来不够“原生”，因此这类应用对用户没有吸引力。

有没有可能结合这两个领域，让原生的动画与控件兼有Web开发的速度？能，得益于iOS的JavaScriptCore引擎，我们可以不用WebView，直接在原生Objective-C代码和JavaScript代码之间搭

建一个桥接层。

React Native是最流行的产品，拥有非常广泛的社区与支持，这让它成为显而易见的选择。然而，记住这项技术仍然处于活跃开发的阶段，因此某些情况下它可能不太稳定。它会出现bug，不过通常很快就会修复。最重要的是，你可以亲手改进几乎所有功能，因为它是一个开源产品。比如，对于某些特殊或者遗漏的原生UI功能，你可以很容易地使用Objective-C/Swift开发自己的原生桥接层。

上一章介绍了一些使用React Native的理由，本章会更进一步。下面列出了与原生Swift/Objective-C开发相比，React Native最重要的优势。

(1) 快速按照设计师原型进行UI开发。React Native使用JSX（类似HTML的语法）以及类似CSS的样式语法，因此即使是设计师也可以通过直接编辑React组件来调整样式、字体以及布局。作为开发者的你更可以轻松调整；并且比起用iOS Storyboard来编辑要快2~3倍。更重要的是，你可以"实时"看到改动显示在移动模拟器/设备上！不用每次都重新编译应用。

(2) 每次修复bug后，无需把应用提交给AppStore审核。你可以很方便地为自己的应用实现一套方案，从远程服务器下载JavaScript文件包，或者使用现有的解决方案，比如CodePush、AppHub.io等。

(3) 组件化开发方式以及使用强大的NodeJS作为依赖管理工具，允许你在分布式团队中开发小型的可复用组件。你可以并行开发多个组件，然后轻松组合起来，这样应用开发的整体速度就显著提高了。

### 4.1.1　npm

每个复杂的应用都包含数十甚至上百个组件，因此需要一个优秀的包管理工具。大量现有的第三方组件能够显著缩短开发时间。React Native支持NodeJS包管理器（npm），这样就可以访问成千上万开发得很棒的流行组件（不过有些模块无法在React Native环境下运行，但是可以在NodeJS中运行）。

### 4.1.2　静态类型检查工具 Flow

Facebook开发的JavaScript静态类型检查工具Flow（https://flowtype.org），允许你为所有数据定义类型，并在输入时检查代码的正确性。

```
function bar(x: string, y: number): string {
  return x.length * y;
}
bar('Hello', 42);
```

以下是运行Flow后（或者在支持Flow的IDE中）的报错信息。

```
3: return x.length * y;
          ^^^^^^^^^^^^ number. This type is incompatible with
```

```
2: function bar(x: string, y: number): string {
                                        ^^^^^^ string
```

总是使用类型声明非常重要，这样可以避免易出错的代码，并且无需运行和测试应用就能及早发现错误。

### 4.1.3　开源

React Native社区有大量的开源组件，并且数量还在不断增长。许多公司把自己的大部分软件都开源了。原因很简单，为了更广泛的社区支持以及更好的软件质量。用开源组件进行应用开发的过程中可能会发现bug，你可以查看源代码并迅速找到问题所在，甚至可以提交修复方案。其他开发者也可以改进你的组件（如果你开源了它）。这样的结果就是，社区的所有成员都是赢家，我们获得了稳定健壮的软件以及数量不断增长的高质量组件。

### 4.1.4　响应式编程

React Native的显著优势之一便是**响应**式编程（reactive programming）。维基百科上写道，“响应式编程是一种编程范式，面向数据流与变动的传播”。这意味着在所用的编程语言中能够轻松表达静态或动态数据流，其背后的执行模型会自动通过数据流传播变动。

例如，在声明式的编程环境中，`a := b + c`是指`a`被赋值为表达式`b + c`立即计算的结果，接下来`b`和`c`的值发生改变，不会影响`a`的值。

然而在响应式编程中，`a`的值会根据新的计算结果自动更新。

React Native允许自定义`View`组件，组合成一棵层次结构树并自动传播变动。这与电子表格很相似，只需把`a`定义成`b`与`c`的和一次，每当`b`或`c`改变时就会更新`a`的值。这样的方式使得代码量更少且更易读，软件质量也比声明式编程好了许多。

### 4.1.5　XMPP

可扩展通讯和表示协议（Extensible Messaging and Presence Protocol，XMPP）是一种基于XML的通信协议，用于面向消息的中间件。它很适合消息通信，并且大部分流行的消息应用都使用它。市面上已经有很多质量卓越的XMPP服务器与客户端软件。我们决定使用processOne提供的ejabberd与开源的iOS客户端库XMPPFramework来开发真正可扩展且稳定的应用。



### 4.1.6　技术栈

我们在应用中使用了以下技术与产品。

- XMPP（ejabberd，iOS XMPPFramework）

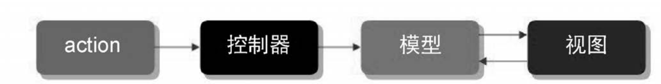
- NPM
- React Native
- 用于开发iOS原生代码的XCode与Objective-C
- Flow
- Mobx（详见第5章）
- Mocha（用于单元测试）
- Bitrise.io用于持续部署

## 4.2 可扩展应用架构

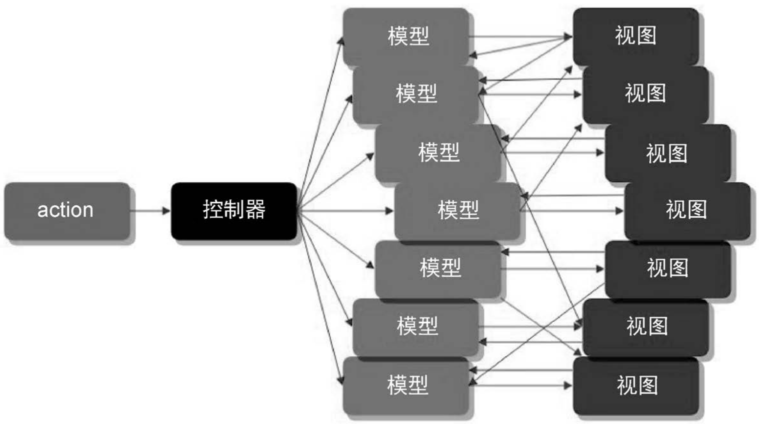
遗憾的是，React Native仅仅是`View`层，没有涵盖一个复杂应用的完整架构。虽然简单的应用可以把全部逻辑都放到React组件中（React Native允许使用`setState`方法来操作状态），复杂的跨平台应用可不能这样做。Android/iOS平台可能需要不同的`View`组件负责渲染一些原生元素，而且React组件要比纯粹的JavaScript更难测试。下面来看看现有的解决方案，从中为我们的应用选择一个最优解决方案。

### 4.2.1 MVC

经典架构模式，模型–视图–控制器。



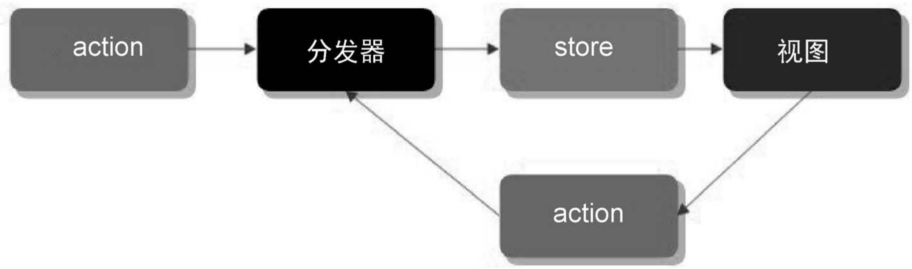

上图的架构看起来非常简洁，但真正复杂应用的情况如下。



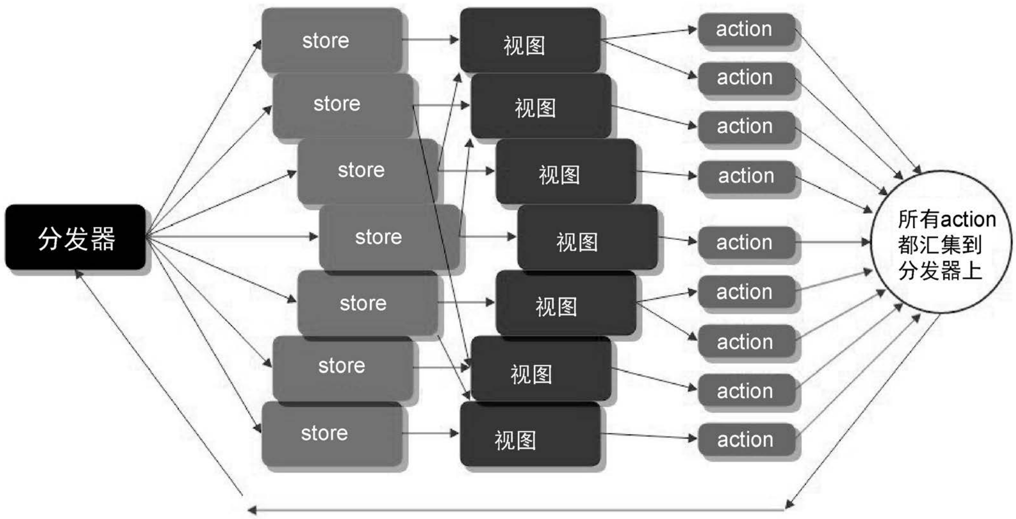
最困难的地方在于，MVC架构不好扩展。由于控制器和视图都可以修改模型，导致难以定位失败的原因，另外修改链很容易就会拉得很长：控制器修改了模型A，模型A再修改视图B，视图B接着修改了模型C，模型C又修改了视图C与D，以此类推。要为复杂应用维护这样的模式非常困难。再者，由于模型和视图全都相互依赖，测试的难度也相当大。

### 4.2.2 Flux

为了改进开发流程，Facebook提出了Flux架构模式来取代MVC。



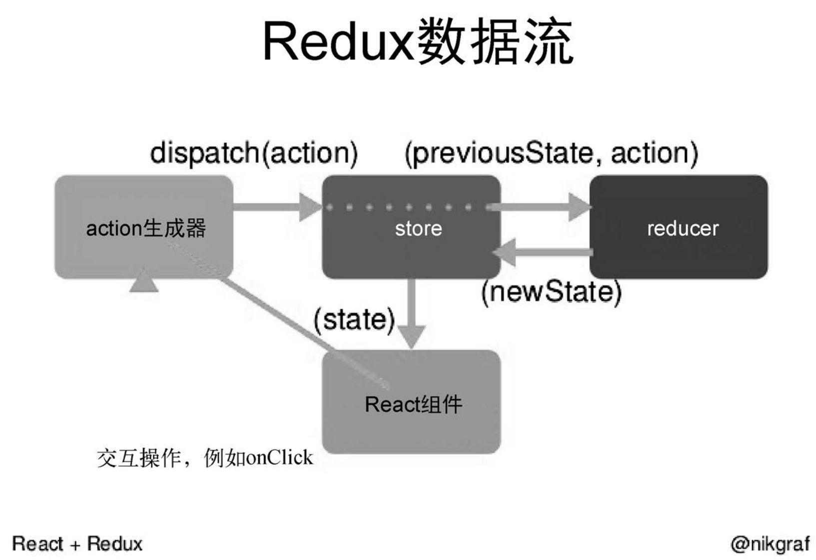

它看起来并没有比MVC好多少，尤其是复杂应用的情况下。




不过，它与MVC有显著的区别，所有数据向一个方向传输（单向数据流），这使得它很方便调试与维护。

### 4.2.3　Redux

Redux（http://redux.js.org/）作为另一个MVC的替代方案，与Flux很相似。它不再需要编写自定义的分发器，而是引入了纯函数形式（pure function）的reducer，按照给定的action对store进行修改，状态改变后视图也随之改变。




由于Redux比MVC和Flux简单，它成为了非常流行的框架。有了Redux的情况下，所有应用状态都表示成一种结构（单一数据源），并且每次action发生之后你都能观察到它是如何改变的。然而，开发者要使用Redux就需要实现很多的模板。

- ❑ 把所有action定义成字符串常量。
- ❑ 按switch/case语句的形式实现reducer，同时生成新的不可变状态。

```
function reducer(state, action) {
    switch (action.type) {
        case ACTION_NAME1:
            return modify1(state)
        case ACTION_NAME2:
            return modify2(state)
        case ACTION_NAME3:
            return modify3(state)
        default:
            return state;
    }
}
```

- 在视图层里触发action。

```
<TouchableOpacity onPress={
    ()=>this.props.dispatch({
        type: ACTION_TYPE,
        data: custom_data
    }>
    <Text>Button</Text>
</TouchableOpacity>
```

如上所示，为reducer传递数据要编写很多冗余的代码：数据映射、繁琐的switch/case语句、action常量。当发起HTTP请求（进行异步操作）时会遇到更多问题，需要使用Redux Thunk（https://github.com/gaearon/redux-thunk）来定义action生成器，参见如下的简化版本。

```
export function fetchPosts(subreddit) {
  return function (dispatch) {
    dispatch(requestPosts(subreddit))
    return fetch(`http://www.reddit.com/r/${subreddit}.json`)
     .then(response => response.json())
     .then(json =>
       dispatch(receivePosts(subreddit, json))
      )
  }
}
```

此外，也可以使用Redux Saga（https://redux-saga.github.io/redux-saga/）或Redux Side Effects（https://github.com/salsita/redux-side-effects）这两个库。

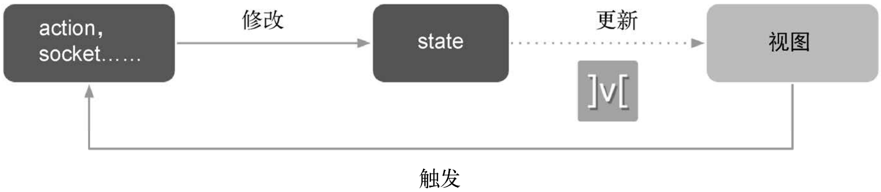
上述代码读写起来不够简单明了。实际上，Redux开发者要亲自编写所有状态管理的代码。Redux在状态管理上几乎已经提供了不少帮助，它只是把action分发给reducer，再把新的状态`connect`到视图上。在使用了Redux几个月后，我们意识到编写模板以及模拟状态转换花费了太多精力。实现一条复杂的异步操作链难度很大，我们的应用包含`Messages`数据，每条`Message`包含参与者（`Profiles`），每个`Profile`又有头像（`File`），`File`又包含元数据（URL和媒体类型），而且所有这些操作都要一个接一个地异步完成。此外，为了高效，还得尽可能缓存数据。应用还有`Friend Lists`数据也包含`Profile`，因此一个好友列表也需要相同的异步操作链。用Redux来完成这样的任务真的是一大挑战。



### 4.2.4 MobX 与 Redux 的比较

直觉上看，寻找一个像React Native一样遵循响应式编程原理的框架意义很大。Flux与Redux首先尝试着模仿了响应性，声明状态（模型）应该如何按照给定的action（事件）来改变。遗憾的是要编写很多代码，并且开发者需要负责保持一切同步。如果某人的个人简介或者头像改变了，状态中任何一处都不会自动随之改变；你需要亲自动手实现。

好在新的框架MobX（https://mobx.js.org/index.html）提供了让所有数据自动同步的特性，你要做的只是用`@observable`来声明数据。它的主要原理与Flux和Redux相似，并且也遵循单向数据流。




对于更复杂的应用来说，很有必要在action和状态之间引入store，好比MVC的控制器或Redux的reducer，也就是改变应用状态的类。每个改变状态的方法应当标记为`@action`。所有依赖属性应当标记为`computed`。当任何所依赖的可观察数据改变时，依赖属性会自动重新计算。

`autorun`与`reaction`函数用于在状态改变时运行用户定义的函数。以下的例子来自我们的应用，对“实时”搜索特性进行处理。

```
reaction(()=> this.global, text => {
  if (!text.length) {
    this.globalResult.clear();
  } else {
    return this.search(text).then(data=> {
      this.globalResult.replace(
         data.hits.map(el=>
           this.profile.create(el.objectID, el)).
               filter(el=>!el.isOwn));
    });
  }
}, false, 500);
```

上述代码对`this.global`变量（输入文本值）的变化进行观察，执行搜索查询（`this.search`函数）并对结果进行处理。参数`500`表示刷新时间（以毫秒为单位），查询操作会在最后一次输入文本值的500毫秒后运行。这样应用就不会在用户输入的过程进行搜索查询，因此UI就更具响应性而且更快。

下面来看看MobX与Redux的主要区别。

(1) Redux的基础是不可变性，而MobX框架是可变的。不可变性本身是很好的概念，也多亏了Redux让它如今这么流行，然而通过这种方式很难实现某些任务。有了可变性以及MobX之后，开发者必须更小心严谨地遵循单向数据流（这样视图就不能更新模型，只能执行store提供的action）。通过这样的方式，就能在不需要任何模板的情况下，很好地处理异步请求与缓存这种很困难的任务。我们的应用使用MobX与领域模型（domain model）下载消息数据，其中包含了消息参与者的ID和消息图片文件的ID，然后下载消息参与者的ID数据，其中包含头像文件的ID，接着下载恰当的文件，执行这些任务的同时还对数据进行高效缓存，并显示先前的消息（这样用户无需等待一切加载完毕，就可以马上开始阅读消息）。相比一大堆的Redux reducer、各种异步操作还有重复数据，MobX的做法更合适。

(2) MobX允许使用类实例（领域对象模型），而Redux只能使用纯JavaScript对象（数组、映射、基本类型）。

(3) MobX提供了计算值（computed value）作为可观察值的简单依赖函数，并自动保持两者同步。在Redux中你需要亲自重新计算这类值并保证结果正确。换句话说，Redux需要你在脑海中模拟所有的状态转换，而MobX会自动为你完成。

(4) MobX包含更精简的代码。它与Redux最主要的区别就是少了各种模板。来看一个简单的计数器应用，它也会计算操作的次数。

### Redux的方式

从actions.js开始：

```
export ACTION_INCREASE = "INCREASE";
export ACTION_DECREASE = "DECREASE";
```

接下来是reducer.js：

```
import {ACTION_INCREASE, ACTION_DECREASE} from 'actions';

export default function reducer(state = {data:0, total:0}, action) {
    switch (action.type) {
        ACTION_INCREASE:
            return { data: state.data + 1, total: state.total + 1 }
        ACTION_DECREASE:
            return { data: state.data - 1, total: state.total + 1 }
        default:
            return state;
    }
}
```

counter.js：

```
import React from 'react';
import { View, Text } from 'react-native';
import { ACTION_INCREASE, ACTION_DECREASE } from 'actions';
import { connect } from 'react-redux';
import Button from 'react-native-button';

function Counter({dispatch, counter, total}){
    return <View>
      <Text>Counter: {counter}</Text>
      <Text>Total clicks: {total}</Text>
      <Button onPress={
          ()=>this.dispatch({type: ACTION_INCREASE}>+
          </Button>
      <Button onPress={
          ()=>this.dispatch({type: ACTION_DECREASE}>-
          </Button>
    </View>;}
export default connect((state) => state)(Counter)
```

4

app.js：

```
import React, { Component } from 'react-native';
import { createStore, Provider } from 'react-redux';
import reducer from './reducer';
import Counter from './counter'
import createStore from './createStore'
const store = createStore(reducer)

const Main = () => {
  return (
    <Provider store={store}>
      <Counter />
    </Provider>
  )
}

export default Main
```

### MobX 的方式

store.js：

```
import {action, reaction, observable} from 'mobx';
import autobind from 'autobind-decorator'

@autobind
class CounterStore {
  @observable counter = 0;
  @observable total = 0;

  @action increase(){
    this.counter++; this.total++;
  }

  @action decrease(){
    this.counter--; this.total++;
  }
}

export default new CounterStore();
```

app.js：

```
import React from 'react';
import { View, Text } from 'react-native';
import { observer } from 'mobx-react/native';
import Button from 'react-native-button';
import store from './store';

@observer
function Counter({store}){
   const store = this.props.store;
   return <View>
      <Text>Counter: {store.counter}</Text>
```

```
      <Text>Total clicks: {store.total}</Text>
      <Button onPress={store.increase}>+</Button>
      <Button onPress={store.decrease}>-</Button>
    </View>
}

const Main = () => {
  return <Counter store={store}/>
}

export default Main
```

如果你在最简单的计数器应用中都看到了差别，那么可以想象，把TinyRobot应用从Redux迁移到MobX后的差别会有多大。以下是用MobX写的Chats模型（聊天列表）的例子。用Redux完成同样的任务要多出30%的代码。

```
export default class Chats {
  @observable _list:[Chat] = [];
  @computed get list(): [Chat] {
    return this._list.sort((a: Chat, b: Chat)=>{
      if (!a.last) return 1;
      if (!b.last) return -1;
      return b.last.time - a.last.time;
    });

  }

  @action add = (chat: Chat): Chat => {
    assert(chat, "chat should be defined");
    console.log("Chats.add", chat.id);
    const existingChat = this.get(chat.id);
    if (!existingChat){
      this._list.push(chat);
    } else {
      console.log("Chat exists");
      return existingChat;
    }
    return chat;
  };

  get(id:string): Chat {
    return this._list.find(el=>el.id === id);
  }

  @action clear = () => {
    this._list.splice(0)
  };

  @action remove = (id: string) => {
    assert(id, "id is not defined");
    this._list.replace(this._list.filter(el=>el.id != id));
  };

}
```

MobX中action成为了真正的JavaScript函数，并且MobX会自动同步可观察模型与视图。

### 4.2.5　领域对象模型

我们决定把所有应用状态都放到一个类里（也就是Redux的“单一数据源”概念），这个类就是包含可观察变量的模型。不同的store改变模型中对应的部分（就像Redux的reducer改变全局应用状态中属于它的那一部分），一旦模型改变，视图就重新渲染。这样一来，就能结合两个领域的概念：MVC中简单的JavaScript语法，以及Redux的可预测性与清晰的状态管理。在MobX中仍然能看到每次action后的任何状态变化（通过logging模型）。

另一项重大决定是在模型中使用对象而非基本类型（Redux只操作基本类型值）。这对于解决之前提到的异步串联请求任务很有必要。另外，这样做也允许使用MobX的“计算”属性，或者根据其他可观察值来自动刷新值。

以下是消息对象的例子。

```
export default class Message {
  id: string;
  @observable from: Profile;
  @observable to: string;
  @observable media: File;
  @observable unread: boolean = true;
  @observable time: Date;
  @observable body: string;
  @observable composing: boolean;
  @observable paused: boolean;
  @computed get date(){ return moment(this.time).calendar()}
```

下面是个人资料类（检查计算值displayName并加载action）。

```
export default class Profile {
  user: string;
  @observable firstName: string;
  @observable lastName: string;
  @observable handle: string;
  @observable avatar: File = null;
  @observable email: string;
  @observable error: string;
  @observable phoneNumber: string;
  @observable location: Location;
  @observable loaded: boolean = false;
  @observable isFollower: boolean = false;
  @observable isFollowed: boolean = false;
  @observable isNew: boolean = false;
  @observable isBlocked: boolean = false;
  @computed get isMutual(): boolean { return this.isFollower && this.isFollowed };

  profile;
  model;
  file;
```

```
@computer get isOwn() {return this.model.profile && this.model.profile.user === this.user}

// model：全局模型
// profile：负责缓存/加载个人资料的ProfileStore
// file：负责缓存/加载文件的FileStore
constructor(model, profile, file, user: string, data) {
  this.model = model;
  this.profile = profile;
  this.file = file;
  this.user = user;

  if (data){
    this.load(data);
  } else {
    when(()=>model && profile && model.connected,
      ()=>this.profile.request(user).then(this.load));
  }
}

@computed get displayName(): string {
  if (this.firstName && this.lastName){
    return this.firstName + " " + this.lastName;
  } else if (this.firstName){
    return this.firstName;
  } else if (this.lastName){
    return this.lastName;
  } else if (this.handle){
    return this.handle;
  } else {
    return ' ';
  }
}

@action load(data){
  this.loaded = true;
  Object.assign(this, data);
  if (data.avatar && (typeof data.avatar === 'string')){
    this.avatar = this.file.create(data.avatar);
  }
}
```

我们使用了控制反转原则（Inversion of Control，https://en.wikipedia.org/wiki/Inversion_of_control）使应用更加模块化与可扩展，领域模型实例接收各自store的实例。如以上代码所示，`Profile`领域模型依赖了`Model`、`ProfileStore`以及`FileStore`，因此它能够从store中加载自己，并创建相关的对象（`File`表示个人资料头像）。之后还可以添加更多逻辑，比如在报错的情况下尝试重新加载。



## 4.2.6　依赖注入

对简单的应用来说，使用NodeJS的`import`语句以及`Singleton`单例模式的store实例就足够了。但这种方式不适合复杂的应用，因为这样不够灵活而且很难测试。另外，这样还会带来某种**隐式**

依赖，就像全局变量依赖那样。

假设有一些模块依赖了XMPP的store（通过NodeJS的`import`语句）。怎样才能把XMPP库从StropheJS切换到iOS的原生解决方案？在完全不涉及XMPP的情况下怎么测试这些模块？最好且最清晰的方式就是直接引入XMPP模块，并通过构造函数传递XMPP实例。这样你就能模拟XMPP模块并传递给所有的单元测试。

然而，复杂的应用通常有很多store要依赖其他store，创建这类store非常麻烦。所幸npm的JavaScript模块`constitute`（https://github.com/justmoon/constitute）提供了自动依赖注入的功能。

原生JavaScript代码的实例化示例如下。

```
function main () {
  const electricity = new Electricity()
  const grinder = new Grinder(electricity)
  const heater = new Heater(electricity)
  const pump = new Pump(heater, electricity)
  const coffeeMaker = new CoffeeMaker(grinder, pump, heater)
  coffeeMaker.brew()
}
```

用`constitute`这样的工具改写后，代码如下。

```
function main () {
  const coffeeMaker = constitute(CoffeeMaker)
  coffeeMaker.brew()
}
```

应用的全局模型`RootStore`包含了其他所有的store。我们用constitute的`@Dependencies`装饰器为每个store声明依赖，最后用`constitute(RootStore)`创建根实例。

为测试代码注入模拟数据，可以像下面这样写。

```
const container = new Container();
container.bindClass(XMPP, MockXMPP);
const mock: RootStore = container.constitute(RootStore);
```

接着模拟数据就会包含所有store与模型，并且内置了模拟的XMPP模块。阅读4.6节了解更多细节。

### 4.2.7　持久化

所有复杂的React Native应用都有一个明显的缺点，即缺少强大的持久化机制。原生iOS应用有CoreData这样的框架，还有很多像MagicalRecord这样的第三方库，能以简单有效的方式来持久化与访问数据。遗憾的是，为React Native开发CoreData的原生桥接层会有问题，因为所有CoreData数据实体以及它们之间的关系需要定义成Objective-C或Swift类。另一种选择就是使用react-native-sqlite-storage（https://github.com/andpor/react-native-sqlite-storage）这样的SQLite桥接

组件，不过它只提供了很底层的API，开发者需要直接编写SQL查询语句。

React Native提供了`AsyncLocalStorage`类作为简单的键值式持久化机制，不过这对于复杂应用来说肯定不够。Redux允许开发者通过`redux-persist`在AsyncLocalStorage中保存或加载应用状态，这样可以从iOS的文件存储中加载初始状态，并在每次action之后保存更新过的状态。这对于简单的应用来说已经足够了，但如果你的状态包含了上百个数据实体，每次action后对全部数据进行保存将会非常低效。

幸运的是，最近发布了React Native组件Realm（https://realm.io/），它提供了CoreData的一切功能，可通过基于数据实体的高级API来简单高效地保存或加载模型。

```
// 定义模型与模型属性
class Car {}
Car.schema = {
  name: 'Car',
  properties: {
    make: 'string',
    model: 'string',
    miles: 'int',
  }
};
class Person {}
Person.schema = {
  name: 'Person',
  properties: {
    name:    {type: 'string'},
    cars:    {type: 'list', objectType: 'Car'},
    picture: {type: 'data', optional: true}, // 可选属性
  }
};

// 获取默认的Realm，并支持我们的对象
let realm = new Realm({schema: [Car, Person]});

// 创建Realm对象并写入本地存储
realm.write(() => {
  let myCar = realm.create('Car', {
    make: 'Honda',
    model: 'Civic',
    miles: 1000,
  });
  myCar.miles += 20; // 更新属性值
});

// 查询Realm，获取所有高里程车辆
let cars = realm.objects('Car').filtered('miles > 1000');

// 返回的结果对象包含1辆车
cars.length // => 1

// 添加另一辆车
realm.write(() => {
  let myCar = realm.create('Car', {
```

4

```
    make: 'Ford',
    model: 'Focus',
    miles: 2000,
  });

// 查询结果实时更新
cars.length // => 2
```

### 4.2.8　应用状态管理

复杂应用最大的问题在于其复杂的状态。开发者通常会忽略这个事实，并且只用布尔标志位（boolean flag）和字符串状态变量来管理应用状态逻辑。举例来说，应用在启动后可以从AsyncLocalStorage中加载自身的状态。这段时间内应用应该在屏幕上显示“加载”界面。应用加载完成后，有用户凭据的情况下应进行登录，否则就显示“宣传”界面。另外，连接成功后得到的个人资料可能不够完善，这时就要把用户重定向到“注册”界面。如果用户的昵称（用户名）已填，他们就会被重定向到登录后的界面（首页）。判断当前应该显示哪个界面变得非常麻烦。

```
function getInitialState(){
    if (this.error || !this.server){
      return "promo";
    }
    if (!this.loaded || this.connecting ||
    this.tryToConnect || (this.connected &&
        this.profile && !this.profile.loaded)){
      return "launch";
    } else {
      return this.profile && this.profile.loaded ?
          this.profile.handle ? "logged" : "signUp" : "promo";
    }
}
```

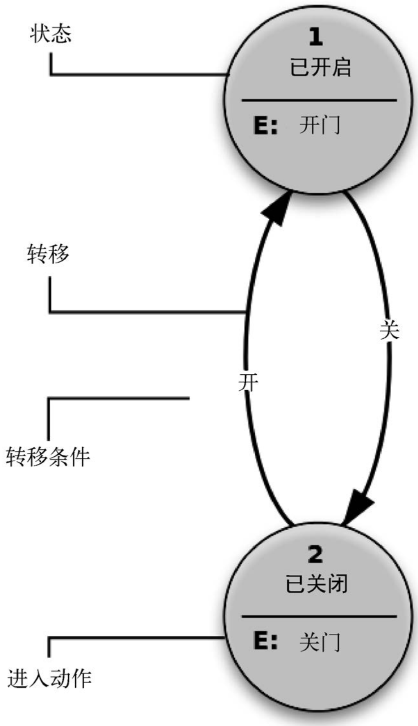
上面的代码相当复杂且无法扩展。比如，如果要在用户登录期间显示另外的UI界面（比如某种进度条动画），这种情况下要怎么修改代码？而且我们无法确定是否遗漏了已有布尔标志位的某些组合，或者是否正确实现了当前界面的判断逻辑。

某些布尔标志位的组合显然没有任何意义，比如this.connecting与this.connected不可能同时为true，this.error与this.connecting同理。因此，当应用连接成功后，开发者也要对这种情况做处理（将this.error设置为null，将this.connecting设置为false，将this.connected设置为true）。如前所述，这样很容易出错，并且会让应用的状态失效。

复杂应用的状态还有另一个问题，即如何理解它并从源码的角度看到它的全貌。“一图胜千言”这句话也适用于应用状态。UML可以解决这个问题，不过这需要开发者付出额外的精力来绘制所有的状态图。

这种情况下，当应用状态变得太过复杂时，很有必要看清它的全貌，此时有限状态机（https://en.wikipedia.org/wiki/Finite-state_machine）对状态管理很有帮助。它是一种用于设计计算

机程序的数学模型。我们可以把应用表示成状态的集合，在不同状态之间进行转移。应用把收到的事件发送给状态机，然后就会根据状态流来自动计算应用状态。这样一来，上述代码就会变成一个仅有的可观察模型`this.state`（它会在收到应用事件之后改变）。以下是一个简单状态机的示例。




编译器、解析器以及词法分析器内都广泛使用了状态机。不过在开始实现应用的状态机之前，先来研究一下这种方法是否完全没有问题。遗憾的是，经典状态机的用途相当有限，而且对复杂应用来说不太可用。它存在以下问题。

(1) 经典状态机的状态不能包含自身数据（模型）。不可能把所有应用状态都描述成状态集合，应用状态包含了大量数据，比如我们的消息应用中就有用户凭据、聊天列表等数据。另外，事件也能包含数据（比如`login`事件的用户凭据数据）。

(2) 不能定义通用的事件处理函数。假设我们需要定义一些`disconnect`处理函数，或者为一些**连接成功**的状态（应用在线时的所有状态）定义类似行为，这样在`disconnected`事件后它们应当转换成`Disconnected`状态。如果使用经典状态机，那么你得为所有**连接成功**的状态定义一个`disconnected`转换行为，使其转换为`Disconnected`状态。然而这种做法不可用，还包含大量冗余

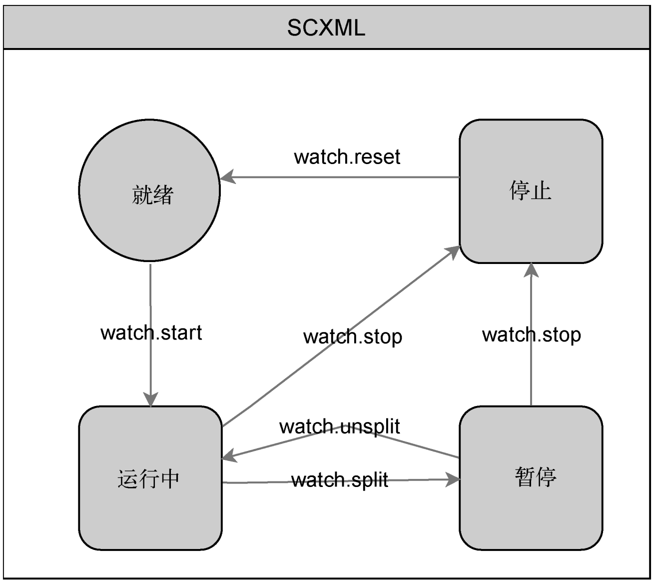
代码。再者，这样还无法扩展。要是以后需要在其他状态下处理连接断开操作该怎么办？分层状态机（hierarchical state machine，HSM）很好地解决了这个问题。它允许定义应用状态的层级，并且祖先层级可以为所有子层级处理某些通用事件（比如前面例子中的`disconnected`事件）。

(3) 经典状态机的另一个问题在于无法同时处理多个状态。假设应用需要接收一条即将到来的消息，并在对其进行处理后添加到模型中。但应用此时可能处于其他状态，并且不能在消息处理过程中离开当前状态。你应该把这类**互不相关**状态的全部有效组合定义成另外的应用状态，这样在经典状态机的情况下状态数量将成倍增加！并行状态很好地解决了这个问题，你可以把某些状态定义成**并行**的，这样它们就可以同时被激活。

(4) 经典状态机的问题还在于它们无法在每次转换后**记住**最初状态（即跟踪状态历史）。假设你想要实现某些错误处理程序，比如显示应用弹窗，记录错误信息并将其发送给服务端，再返回到应用最初的状态。用经典状态机该怎么解决这种情况？你需要为每个应用状态定义独立的“错误处理”状态。

20世纪80年代，David Harel提出的**状态图**（statechart，现已成为UML规范的一部分）解决了上述问题。状态图可扩展标记语言（State Chart extensible Markup Language，SCXML，参考网址为https://www.w3.org/TR/scxml/）是一门XML语言，它以Harel状态图为基础，提供了基于通用状态机的执行环境。它的标准经过W3C认证，并允许用XML文件来描述全部状态。SCXML非常灵活，可以像定义条件转换过程一样定义复合与并行的状态(因此已登录状态就负责处理连接断开事件，并把断开错误信息显示给用户，所有已登录状态下显示的UI界面可以继承它，并且无需关心该事件)。每个状态可以拥有可执行的`onEntry`与`onExit`，转换过程也能拥有这种自定义action。

以下状态图描述了秒表的行为。



描述此图中转换过程的SCXML文件如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       version="1.0" initial="ready">
    <state id="ready">
        <transition event="watch.start" target="running"/>
    </state>
    <state id="running">
        <transition event="watch.split" target="paused"/>
        <transition event="watch.stop" target="stopped"/>
    </state>
    <state id="paused">
        <transition event="watch.unsplit" target="running"/>
        <transition event="watch.stop" target="stopped"/>
    </state>
    <state id="stopped">
        <transition event="watch.reset" target="ready"/>
    </state>
</scxml>
```

（Apache许可，请参见网页 http://commons.apache.org/proper/commons-scxml/usecases/scxml-stopwatch.html。）

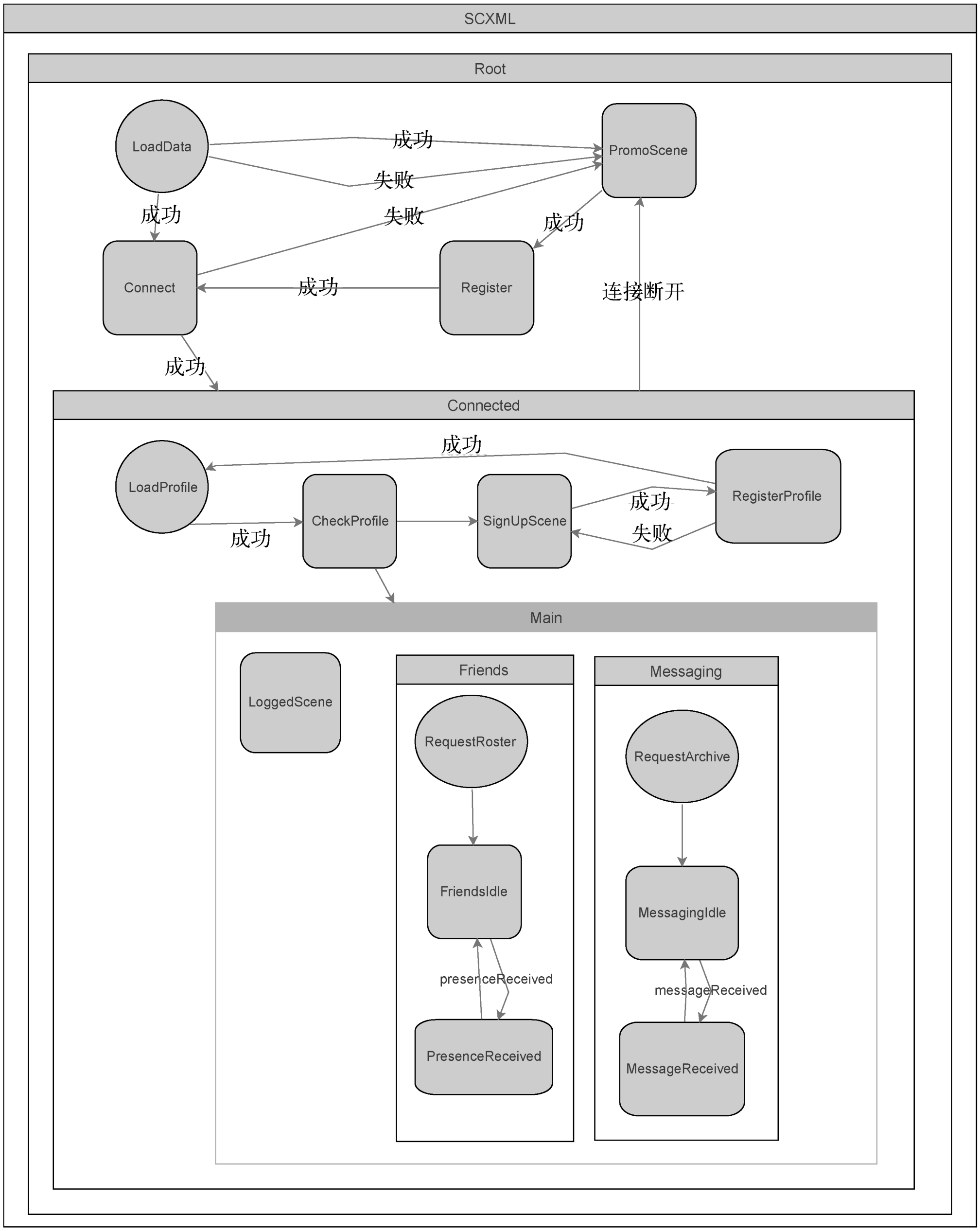
有很多JavaScript状态机框架，最出名的就是Machina.js（http://machina-js.org/）与JavaScript State Machine（已过时，网址为http://codeincomplete.com/posts/javascript-state-machine-v2-3-0/）。`machina-js`支持分层状态，但不支持并行与历史记录（不支持SCXML）。另外，它也不能可视化状态流（显示为状态图）。

幸运的是，SCXML有JavaScript的实现，即scxml（SCION，网址为https://www.npmjs.com/package/scxml）。SCION 2.0是一个轻量级SCXML转JavaScript编译器，面向以SCION为核心的状态图解析器。它目前只支持Node.js和浏览器，未来会提供对Rhino以及其他JavaScript环境的支持。

SCION自动生成的代码不是ES6风格，而且没有Flow的严格类型检查注释，因此我们开发了Statem（https://github.com/aksonov/statem）：由ES6 + Flow + MobX驱动的代码生成器，让你可以轻松访问所有状态（通过简单的`statem.yourState`语句），而且状态是完全响应式的。

我们使用SCXML的GUI编辑器scxmlgui（https://github.com/fmorbini/scxmlgui）可视化和编辑应用的状态图。它能够操作SCXML文件，并允许导出成SVG/PNG/DOT等多种格式。当然你也可以手动编辑SCXML文件，因为它的结构非常简单明了。

以下是我们的应用状态简化版（由scxmlgui编辑）。




如图，你可以看到复合状态（`Root`、`Connected`和`Main`）以及并行状态（`LoggedScene`、`Friends`和`Messaging`）。

不过，怎样才能把应用逻辑与这张状态图进行连接呢？来看看以下的SCXML文件。

```
<scxml name="TinyRobot" version="1.0" xmlns="http://www.w3.org/2005/07/
scxml">
  <datamodel>
    <data expr="this.sm.storage" id="storage"/>
    <data expr="this.sm.xmpp" id="xmppStore"/>
    <data expr="this.sm.friend" id="friendStore"/>
    <data expr="this.sm.profile" id="profileStore"/>
    <data expr="this.sm.message" id="messageStore"/>
    <data expr="this.sm.model" id="model"/>
  </datamodel>
  <state id="Root" initial="LoadData">
    <state id="LoadData">
      <onentry>
        <promise>storage.load()</promise>
      </onentry>
      <transition cond="_event.data
         &amp;&amp;_event.data.user" event="success" target="Connect"/>
      <transition cond="!_event.data ||
        !_event.data.user" event="success" target="PromoScene"/>
      <transition event="failure" target="PromoScene"/>
    </state>
    <state id="PromoScene">
      <transition event="success" target="Register"/>
    </state>
    <state id="Register">
      <onentry>
        <promise>xmppStore.register(_event.data.resource,
            _event.data.provider_data)</promise>
      </onentry>
      <transition event="success" target="Connect"/>
    </state>
    <state id="Connected" initial="LoadProfile">
      <onentry>
        <on event="disconnect">xmppStore.disconnected</on>
      </onentry>
      <transition event="disconnect" target="PromoScene"/>
      <state id="CheckProfile">
        <onentry>
          <assign expr="_event.data" location="model.profile"/>
        </onentry>
        <transition cond="!this.model.profile.handle" target="SignUpScene"/>
        <transition cond="this.model.profile.handle" target="Main"/>
      </state>
      <state id="SignUpScene">
        <transition event="success" target="RegisterProfile"/>
      </state>
      <state id="RegisterProfile">
        <onentry>
          <promise>xmppStore.update(_event.data)</promise>
        </onentry>
        <transition event="failure" target="SignUpScene"/>
        <transition event="success" target="LoadProfile"/>
      </state>
      <parallel id="Main">
```

4

```
        <state id="LoggedScene"/>
        <state id="Messaging" initial="RequestArchive">
          <state id="RequestArchive">
            <onentry>
              <on event="messageReceived">xmppStore.message</on>
              <script>messageStore.requestArchive()</script>
            </onentry>
            <transition target="MessagingIdle"/>
          </state>
          <state id="MessagingIdle">
            <transition event="messageReceived" target="MessageReceived"/>
          </state>
          <state id="MessageReceived">
            <onentry>
              <script>messageStore.receiveMessage(_event.data)</script>
            </onentry>
            <transition target="MessagingIdle"/>
          </state>
        </state>
        <state id="Friends" initial="RequestRoster">
          <state id="RequestRoster">
            <onentry>
              <promise>friendStore.requestRoster()</promise>
            </onentry>
            <transition target="FriendsIdle"/>
          </state>
          <state id="FriendsIdle">
            <transition event="presenceReceived" target="PresenceReceived"/>
          </state>
          <state id="PresenceReceived">
            <transition target="FriendsIdle"/>
          </state>
        </state>
      </parallel>
      <state id="LoadProfile">
        <onentry>
          <assign expr="_event.data.host" location="model.server"/>
          <promise>profileStore.loadProfile(_event.data.user)</promise>
        </onentry>
        <transition event="success" target="CheckProfile"/>
      </state>
    </state>
    <state id="Connect">
      <onentry>
        <promise>xmppStore.connect(_event.data.user,
            _event.data.password,
            _event.data.host)</promise>
        <assign expr="_event.data.host" location="model.server"/>
      </onentry>
      <transition event="failure" target="PromoScene"/>
      <transition event="success" target="Connected"/>
    </state>
  </state>
</scxml>
```

- this.sm引用了StateMachine实例。你可在此处传入所有store或数据，以及自定义action。
- _event.data是SCXML的内置变量，包含了转换参数。
- promise和on是Statem提供的SCXML扩展。它们允许你使用JavaScript的promise，并根据promise的结果生成success或failure事件。你也可以自行定义任何其他的action。

如何才能把SCXML集成到React Native应用中呢？

- 执行npm i statem --save，安装statem。
- 执行npm i watchman --save，安装watchman。
- 把model.scxml文件放到项目中的state文件夹下，并在package.json的scripts部分添加以下代码。

```
"watch": "./node_modules/watchman/watchman
        state/model.scxml 'npm run gen'",
"gen": "node node_modules/statem/src/parser.js
         state/model.scxml gen/state.js",
```

之后运行npm run watch。

- 在App.js中导入生成的genstate.js，创建状态并传入所有store以及可选的自定义action。

```
import createState from '../gen/state';
const statem = createState({...rootStore, ...customactions});
```

- 如果不想在SCXML文件内调用JavaScript代码，也可以按照如下方式为每个状态定义onEntry和onExit方法。

```
statem.stateId.onEntry = (_event)=>{...}
```

需要注意的是，所有状态的ID应当是独一无二的，并且要以大写字母开头（好比JavaScript的类），不要包含空格以及其他特殊字符（也就是说，须为有效的JavaScript标识符）。statem会把它们全都添加到自己的实例中（实例以小写字母开头）。因此，如果你有一个名为Register的状态ID，可以通过statem.register访问这个状态的数据。你也可以通过import {RegisterState} from '../gen/state'来使用注册的生成类，这样可以用Flow进行严格类型检查。

- 把statem传给Router（如果用到了RNRF组件），或者直接传给React Native组件。

```
<Router statem={statem}>
...
</Router>
```

- 在组件内，可以像调用普通的JavaScript方法一样，使用所有的状态转换以及其他数据。

```
statem.success({resource: DeviceInfo.getUniqueID(), provider_data})
```

SMC（http://smc.sourceforge.net/）是另一个SCXML的替代方案，它按自己的格式描述应用状态流，并且可以为项目生成JavaScript代码。我们选择SCXML，因为它是W3C认证的标准。

4

### 4.2.9　设计模式

如果想要开发很复杂的应用，并希望方便长时间维护，这种情况下把所有代码拆分成小而清晰的功能块就显得尤为重要了。每个类只应负责单一类型的操作。了解设计模式对于正确拆分应用非常重要。我们的应用包含了以下这些重要的设计模式。

- **观察者模式**（响应式模型）
- **工厂方法模式**（创建store的领域对象）
- **对象池模式**（缓存store的领域对象来节省流量与时间）
- **单例模式**（store与模型）
- **控制反转模式**（用npm提供的`constitute`组件来管理store与模型的依赖）
- **备忘录模式**（用iOS的本地存储或Realm持久化应用状态）
- **活跃记录模式**用于表示应用的Realm模型实体
- **策略模式**（根据环境使用不同的XMPP连接算法）
- **门面模式**（用于简化对底层导航组件的访问，详见第5章）
- 异步xmpp操作中大量使用了promise以及`async/await`

### 4.2.10　应用架构

TinyRobot应用的整体架构如下所示。

- model/*：可观察且响应式的MobX领域模型类（`Model`）。
- model/Model.js：顶级模型，表示全局应用状态。
- stores/*：store代表应用的业务逻辑，通常是一系列可以修改`Model`的action集合。
- stores/RootStore.js：顶级store，包含其他所有store和顶级`Model`。
- components/*：React Native组件（`View`，视图），观察模型变化并相应地重新渲染自身。
- App.js：把一切组合起来的根文件，定义所有导航界面（请阅读第5章了解更多信息）并把`RootStore`实例传递给它们（npm模块`constitute`用来方便地管理store依赖并创建store的单例）。

## 4.3　导航

导航在任何移动应用中都是非常重要的一部分。复杂的应用会密集地使用导航，并且会包含数十个UI“界面”（scene），因此正确的选择非常重要。React Native提供了一些内置的组件：`NavigatorIOS`、`Navigator`以及`ExperimentalNavigation`。另外本节也会对几个最流行的第三方导航组件进行评价。

### 4.3.1 NavigatorIOS

它对iOS原生的UINavigationController做了一层封装。它的扩展性很有限，而且React Native团队不提供支持。然而，它可以为应用提供原生的过渡动画和体验，因此它适合那些不需要自定义选项以及复杂导航的小型应用。

#### 1. Navigator

跨平台的纯JavaScript实现方案拥有更多的自定义选项，但缺乏原生的过渡“体验”，开发者必须调整过渡动画，使其看起来和原生的很接近。它的API非常简单，比如push/pop/reset方法。然而它没有完全面向“界面”，也就是说开发者必须在push方法内定义所有的界面属性。

#### 2. ExNavigator

ExponentJS团队开发的第三方组件，允许开发者在独立的类中定义“界面”，而且导航代码比Navigator和NavigatorIOS的更清晰。

#### 3. ExperimentalNavigation

前面几个组件的显著缺点就是命令式的API以及状态化机制，所有导航状态都保存在这些组件中，状态的访问和测试都很困难。React Native团队开发的最新ExperimentalNavigation API引入了类似Redux的做法，所有导航状态都保存在一个不可变的变量中，导航action通过对应的reducer改变状态。导航状态变化后，React Native组件就会相应地改变导航界面。

#### 4. React Native Router Flux

上述所有组件的API都各不相同，因此开发者无法在不重写大量代码的前提下在它们之间方便地切换。一年多以前，只有Navigator和NavigatorIOS可以用，而且它们的API不好用。随着时间的推移肯定会出现更好的导航组件，因此Pavlo Aksonov通过实现**门面模式**（Facade Design Pattern）开发了React Native Router Flux（RNRF）。这个包在已有的导航组件上增加了额外的抽象层，以便使用统一的语言来描述应用界面。RNRF包的第一版（v1.*x*）使用了Navigator，第二版使用了exNavigator，第三版使用了最新的NavigationExperimental。

iOS Storyboard的概念展示了一种绝佳的可能性，可以在一个地方查看所有应用界面，并且它很适合理解应用架构。RNRF包也尝试实现同样的目的。导航状态可以表示成树状结构，因此最佳选择就是使用JSX语法。

以下示例展示了带有自定义导航栏与标签栏的三个界面。

```
<Router navBar={NavBar}>
    <Scene key="first" component={First} hideNavBar />
    <Scene key="second" tabs={true}>
        <Scene key="tab1" component={Tab1} tabIcon={TabIcon}/>
        <Scene key="tab2" component={Tab2} tabIcon={TabIcon} rightButton=
{RightButton}/>
    </Scene
</Router>
```

首先，整个界面隐藏了导航栏，并显示包含两个标签的标签栏。第二个标签还包含了右侧导航按钮界面的具体内容（`RightButton`组件）。定义完整个界面后，就可以调用`Action.tab1()`或`Actions.tab2()`从第一个界面切换到第二或者第三个界面。

TinyRobot应用使用了React Native Router Flux包来管理导航状态，因为它允许使用统一的语言来描述应用界面以及路由，不依赖底层的导航组件。RNRF还内置了模态弹窗的支持（语法一致，比如用`Actions.privacyNotes()`显示模态窗，再用`Actions.pop()`隐藏，这样应用代码就不需要了解实际使用了哪种弹窗组件）。该组件还可以很方便地集成侧边菜单这种自定义导航容器。

### 4.3.2　注册与认证流程

几乎所有应用都使用典型的初始导航流程，即启动应用界面等待所有数据加载完毕，如果不存在（已登录）用户就前往注册/登录界面。这种实现过程有些麻烦，并且有多种实现方式。许多开发者通过命令方式来实现，在初始组件的`componentDidMount`方法内检查应用状态，然后调用`Actions.login()`或者`Actions.home()`这样的方法（也可以push `Navigator`的API）。不过，很有必要让代码**延迟**执行，以便让React Native完全渲染UI。

```
componentDidMount(){
    if (!this.props.isLoggedIn){
        setTimeout(()=>Actions.login());
    } else {
        setTimeout(()=>Actions.home());
    }
}
```

上面的代码有些糟糕，调用`setTimeout`显得不够优雅；然而更大的问题在于，业务逻辑放到了UI里面！你需要为其他设备的UI组件复制这段逻辑（同样的界面在Android和iOS上可能会不同）。这段代码属于典型的业务逻辑部分，必须放在store当中。但store不是React组件，那么要怎样在store里调用React的API呢？

`NavigationExperimental`组件的API试图解决这个问题，把管理导航状态的代码从React组件中抽离。然而，在React Native 0.28之前的版本中，它的`onNavigate`方法仍然和React Native耦合，因此不可能在React之外的store中使用它。最近的一些改动使其变得可能，不过这项技术仍然处于实验性阶段。

相比之下，RNRF包提供了**响应式**的`Switch`界面，以内部的`TabBar`容器为基础，根据用户定义的`selector`函数来切换标签。这样就可以把Redux或MobX管理的应用状态与`Switch`界面进行`connect`，接着在模型或store内定义业务逻辑。

Model.js的代码如下。

```
@computed get scene(): string {
    return this.currentState.name;
}
```

如你所见，这里的逻辑相当复杂：4个UI界面对应不同的应用状态，并把初始应用界面定义成MobX的`computed`值。这样允许你考虑到大量不同的场景（无网络连接、数据过期、注册信息错误或不完整等）。

App.js中RNRF包的代码如下所示。

```
<Router>
    <Scene key="root" component={Switch} tabs={true} selector={({model})=>model.scene}>
          <Scene key="launch" component={Launch} hideNavBar/>
          <Scene key="promo" component={Promo} hideNavBar/>
          <Scene key="signUp" component={SignUp} hideNavBar/>
          <Scene key="logged" component={Drawer} open={false} SideMenu={SideMenu} >
                 ....
          </Scene>
    </Scene>
</Router>
```

这是完全声明式的做法，不需要告诉应用**如何**控制流程（`if/then/else`），只需描述要完成**什么**。代码变得更加简洁而且可靠。举例来说，对于认证失败，会话过期或者注销等情况，不要想着进行`redirect`导航操作。因为一旦store修改了`model.error`或`model.connected`状态，`Switch`与响应式的`Model`就会自动完成这个过程。

### 4.3.3 完美的导航

最新版的RNRF包仍然有一些缺陷。它重度依赖React Native，而且不允许从业务逻辑内调用导航action（React所提倡的不可知论）。因此现在不能在store中调用`Actions.login()`了。另外，因为`Actions`方法由执行环境动态生成，所以无法执行严格类型检查，并且任何IDE都不会为此提供自动补全功能的支持，于是就很容易写错action的名称，比如把`Actions.signUp`写成`Actions.signup`，而且只有在执行环境中才会发现这个错误。RNRF的`Scene`组件不再有固定的`propTypes`集合，它直接把所有属性传给底层的`Scene.component`实例。最终表明React无法验证传给`Scene`组件的属性的正确性。

如此看来，优秀的导航组件应该具备以下两个特点。

(1) 把action的定义分成两部分：业务逻辑，包含清晰导航API的层次树（独立于React Native）。

(2) 在`View`层定义action与视图的映射（响应式）。

要描述包括导航的通用应用流程，Harel状态图再合适不过。因此，理想的导航组件会根据当前应用状态创建`navigationState`（由`ExperimentalNavigation`的API提供支持），并在每次应用状态改变后使用`statem.allStates`（返回应用所有激活的状态，包括祖先状态）自动改变它。分层状态表示常见的Navigation栈，并行状态表示标签式视图（RNRF内的tabs组件）。

因此我们建议的语法如下所示。

```
import createState from './gen/state';
```

```
import {Modal, Scene, Router} from 'react-native-router-flux';
import {Launch} from './Launch';
import {ProfileDetails} from './ProfileDetails';
import {Login} from './Login';
import rootStore from '../store/RootStore';

export default class App extends React.Component {
  constructor(props){
    super(props);
    this.statem = createState({...rootStore});
    this.statem.start();
  }
  render(){
    return (
      <Router statem={statem} >
        <Scene state={statem.modal} component={Modal}/>
        <Scene state={statem.profileDetail}
          component={item => <ProfileDetails item={item} />
          title={item=>item.displayName} />
        <Scene state={statem.launch} component={Launch} hideNavBar/>
        <Scene state={statem.login} component={Login} hideNavBar/>
       // ……其他
      </Router>
    );
  }
}
```

如你所见，React Native的代码内没有树状（分层）结构（使用当前v3版本的RNRF），只是把State映射到View。Scene拥有固定的propTypes集合（key、component、title、hideNavBar、hideTabBar和navBar等）。这意味着所有界面有特定的UI元素，但不包括特定的组件属性。它们通过component函数这种自然而然的形式来定义，并拥有自动补全和严格类型检查等特性。

这一步之后，就可以在Login视图内调用this.props.state.login(username, password)这样的方法了。它会从当前状态发起login转换过程。应用状态会根据状态流程而改变，接着React Router会自动重新构造导航树并渲染下一个界面。

## 4.4　通信

上文提到，我们选择了XMPP作为应用通信技术。遗憾的是，React Native没有为XMPP提供任何支持。因此，需要从现有的iOS原生组件或者基于浏览器的方案中寻找解决办法。Robbie Hanson开发了优秀的iOS框架XMPPFramework，另外Pavlo Aksonov开发的react-native-xmpp可以提供React Native与iOS框架之间的“桥接器”。

另一个方案便是采用流行的Web XMPP库StropheJS（http://strophe.im/strophejs/）。不巧的是，这个库重度依赖浏览器的DOM，而React Native不具备DOM。不过，可以在NodeJS环境下使用NPM包xmldom（https://github.com/jindw/xmldom）来模拟DOM。经过一些调整和修补后，StropheJS

可以在React Native环境中运行。需要注意的是，通信过程由WebSockets（React Native原生支持）完成，因为StropheJS基于Web，不像XMPPFramework那样可以直接连接。

### 4.4.1 原生 vs. JavaScript

选择最合适的方案非常重要，因此需要为两种方案进行性能测试：`react-native-xmpp`与`StropheJS`。我们发现，在iPhone 5s上，JavaScript方案比iOS原生桥接器慢了2~3倍。JavaScript单线程运行，而XMPPFramework利用了多线程。因此TinyRobot采用了原生方案。不过，在NodeJS单元测试中使用了StropheJS（原生方案不能在NodeJS环境中运行），此外我们还开发了通用的XMPP接口，原生和JavaScript方案都遵循这个接口。我们还利用了很实用的StropheJS节（stanza）构造器：`$iq`、`$message`以及`$presence`，用于创建必需的XMPP节。显然，这些节在传递给原生iOS XMPP框架之前会被序列化成字符串。

### 4.4.2 函数式编程

怎样才能把回调式API与应用的响应式代码、观察者变量以及可观察变量相结合？我们应当极力避免“回调地狱”，这种使用回调函数的糟糕代码不够清晰，而且维护、测试和扩展都非常困难。

函数式编程把任何数据源都表示成事件流。这样我们的应用就能把所有的XMPP事件表示成事件源。创建事件流之后，应用就能进行各种转换操作，比如把XMPP的XML节`<message>`转换成JSON，再转换成Message领域对象。TinyRobot使用了KefirJS（https://rpominov.github.io/kefir/）来实现事件流。

```
const message = Kefir.stream(emitter =>
    this.provider.onMessage =
        message => emitter.emit(message)).log('message');

const iq = Kefir.stream(emitter =>
    this.provider.onIQ =
        iq => emitter.emit(iq)).log('iq');
```

此处的`this.provider`就是StropheJS或者iOS的XMPP实现方案。

XMPP的`<iq>`节表示典型的请求/响应场景。举例来说，应用发送请求获取最新的好友列表（名单），服务端返回的`iq`节`response`就会包含好友列表。这个过程可以通过JavaScript Promise完成，同时过滤事件流（数据参数为StropheJS的`$iq`方法创建的`iq`节实例）。

```
sendIQ(data) {
  const id = data.tree().getAttribute('id');
  return new Promise((resolve, reject)=> {
    const stream = this.iq.filter(stanza => stanza.id == id);
    const callback = stanza => {
      stream.offValue(callback);
```

```
        if (!stanza || stanza.type === 'error'){
          reject(stanza ? stanza.error : {text: 'error'});
        } else {
          resolve(stanza);
        }
      };
      stream.onValue(callback);
      this.provider.sendIQ(data);
    });
  }
```

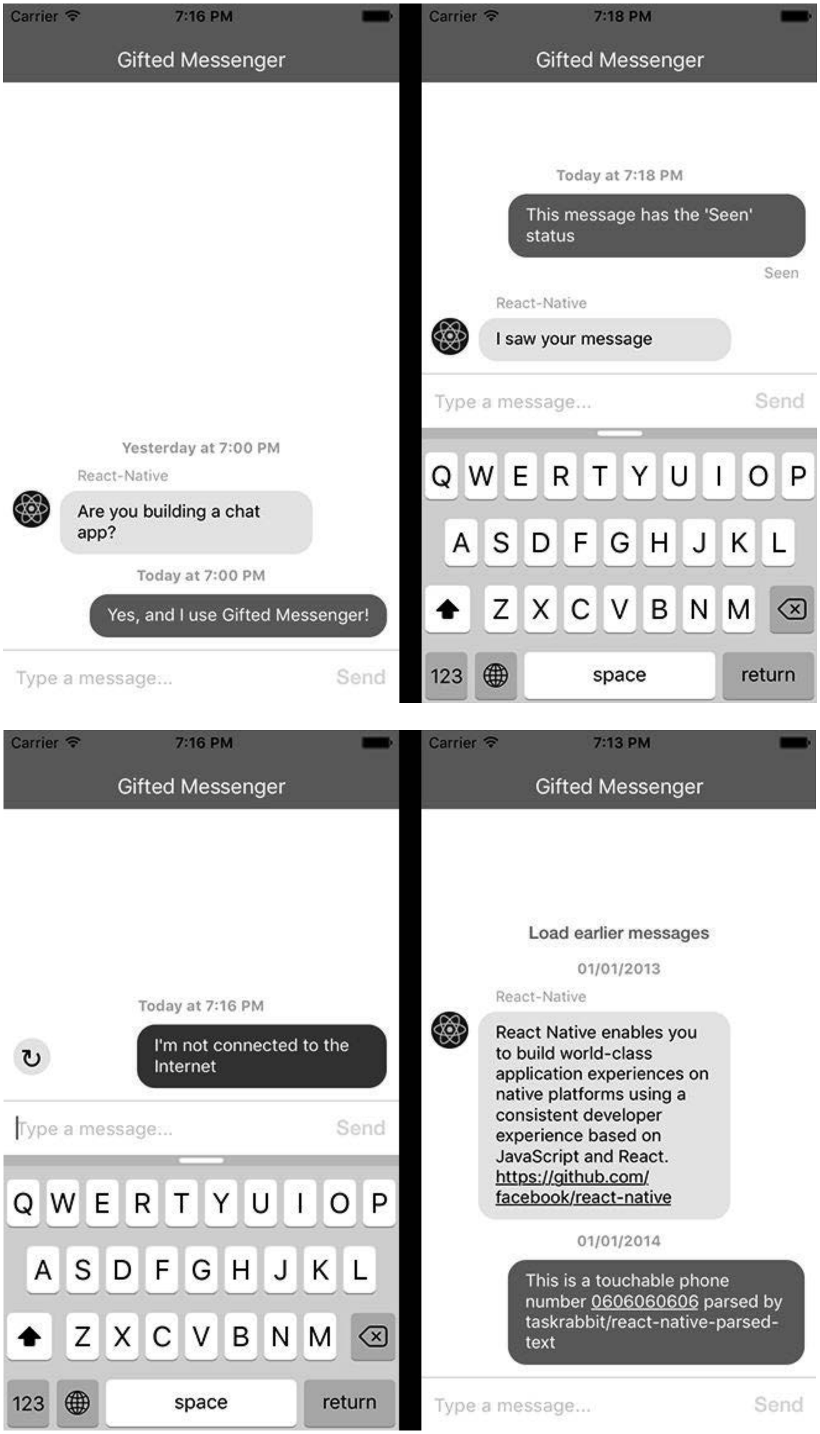
接下来可以使用async和await来请求好友名单。

```
@action async requestRoster(){
  const iq = $iq({type: 'get', to: this.model.profile.user + '@'
      + this.model.server}).c('query', {xmlns: NS});
  try {
    const stanza = await this.xmpp.sendIQ(iq);
    // 处理stanza
    let children = stanza.query.item;
    if (children && !Array.isArray(children)) {
      children = [children];
    }
    // 处理children，把它们添加到模型里
    this.model.friends.add(this.process(children));
} catch (error){
      // 处理错误信息
}
```

如你所见，上述代码非常清晰，不包含任何回调函数。然而，这还不完全是函数式和声明式的代码，因为this.model.friends.add是命令式的代码。这样做的原因在于this.model.friends是可持久化的模型，它有不同的数据源（XMPP/iOS storage/UI）。当然，可以尝试利用merge、mapLatest等函数式操作符进行重构，但这样做会导致代码变得不够清晰且更难维护。在命令式与声明式的编程方式之间找到正确的平衡非常重要；核心的函数式程序很难阅读，这也是引入async与命令式await的原因。它们都可以用函数式的写法代替（promise），但会使代码的可读性变差。

### 4.4.3　用户界面

任何消息应用中最复杂的部分莫过于实现消息对话框的UI。其中存在诸多挑战：加载先前的消息、发送消息失败时进行重试、灵活的日期显示（临近日期的消息无需显示具体日期）、接收到新消息时自动滚动对话框，等等。调研了已有的解决方案之后，我们决定使用第三方开源组件react-native-gifted-messenger（https://github.com/FaridSafi/react-native-gifted-chat）来展现消息UI。


Carrier
7:16 PM
Gifted Messenger
Yesterday at 7:00 PM
React-Native
Are you building a chat app?
Today at 7:00 PM
Yes, and I use Gifted Messenger!
Type a message...
Send
Carrier
7:18 PM
Gifted Messenger
Today at 7:18 PM
This message has the 'Seen' status
Seen
React-Native
I saw your message
Type a message...
Send
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
return
Carrier
7:16 PM
Gifted Messenger
Today at 7:16 PM
I'm not connected to the Internet
Type a message...
Send
Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M
123
space
return
Carrier
7:13 PM
Gifted Messenger
Load earlier messages
01/01/2013
React-Native
React Native enables you to build world-class application experiences on native platforms using a consistent developer experience based on JavaScript and React. https://github.com/facebook/react-native
01/01/2014
This is a touchable phone number 0606060606 parsed by taskrabbit/react-native-parsed-text
Type a message...
Send

我们需要提交一些PR来加强它的定制化，以便为所有UI元素提供我们自己的UI设计。这些元素包括消息“气泡”、文本输入元素以及用户头像，不过这总比从头开发一个这样的组件好多了。更重要的是，开发者能够改善已有的组件，而不是重写一个，这样人人都能受益。

## 4.5　位置

基于位置的应用很大程度上利用了设备的GPS来获取位置数据以及在用户之间共享位置数据。React Native提供了内置的全局对象`navigator`，用于访问用户的位置数据。

```
observe(){
  this.stop();
  if (typeof navigator !== 'undefined'){
    this.watch = navigator.geolocation.watchPosition(
            this.updatePosition, ()=>{},
          {
              enableHighAccuracy: true,
              timeout: 20000,
              maximumAge: 1000
          }
    );
  }
}

@action updatePosition(position) {
    if (this.model.profile){
          this.model.profile.location =
              new Location(position.coords);
    }
}
```

注意，有必要检查`typeof navigator !== 'undefined'`，避免单元测试过程（NodeJS环境）报错。

### UI

在React Native中显示地图，有三个最常用的选择：`react-native-maps`组件（https://github.com/airbnb/react-native-maps）提供的iOS内置地图、Google Maps，以及MapBox（https://www.mapbox.com/）。Google Maps对React Native的集成不够成熟（再者ReactNativeGoogleMaps组件早就过时了，请参考网页https://github.com/peterprokop/ReactNativeGoogleMaps）。MapBox通过React Native MapBox GL（https://github.com/mapbox/react-native-mapbox-gl）提供了对React Native的**实验性**支持。我们决定使用MapBox，因为与原生iOS地图相比，它的定制化程度更高，不过以后可能会发生变化。恰当地设计应用很重要，以便在必要时能够替换地图提供商（需要应用**策略**设计模式）。

## 4.6 部署与单元测试

应用稳定且能正常工作，这一点十分重要。另一个问题在于，你有时需要修改应用，并在必要时进行代码重构（重复代码移除、代码拆分等）。没有自动化单元测试的情况下不可能进行这项工作。在类Flux的单向数据流架构下，这样做非常简单，可以按照以下方式来测试每个action。

(1) 定义你想要的全局模型的初始状态（在`before`代码块内）。

(2) 执行必要的store action。

(3) 把全局模型转换成纯JavaScript结构（如果使用领域模型对象），将其与期望的输出进行比较（期望发生改变的action）。你可能想要检查改变的模型值，但这种情况下你无法确定action没有修改期望之外的其他东西。

异步action的情况会更加复杂。这种情况下你无法进行第3步。幸运的是，MobX的`when`函数可以很容易做到这一点（这里使用`Mocha`作为测试容器，`step`插件用来进行有序测试）。

```
step("register/login", function(done){
  const register = testDataNew(11);
  profile.register(register.resource, register.provider_data);
  when(()=>model.profile && model.connected && model.server, done);
});
```

上述代码非常简洁而且易读。`when`函数在第一个函数参数返回`true`的情况下会执行第二个函数参数（也就是响应式模型相应地发生变化时）。如果需要对模型进行额外的检查，可以在调用`done()`之前进行。

```
when(()=>model.profile && model.connected && model.server,
    ()=>{
        // 在此处验证模型是否正确
        done();
    });
```

另外，要始终牢记单元测试在NodeJS环境中运行，而不是在React Native中运行。因此你需要模拟用到的所有React Native的特定组件，或者检查`typeof(nativeComponent) === undefined`以避免报错。

4

### 4.6.1 React Native 组件测试

与测试React组件的方法类似，只需传入一些数据作为组件属性，接着观察它如何渲染。AirBnb开发的enzyme（http://airbnb.io/enzyme/）可以很方便地测试React Native组件。

```
import React from 'react';
import { mount, shallow } from 'enzyme';

describe('<Foo />', () => {

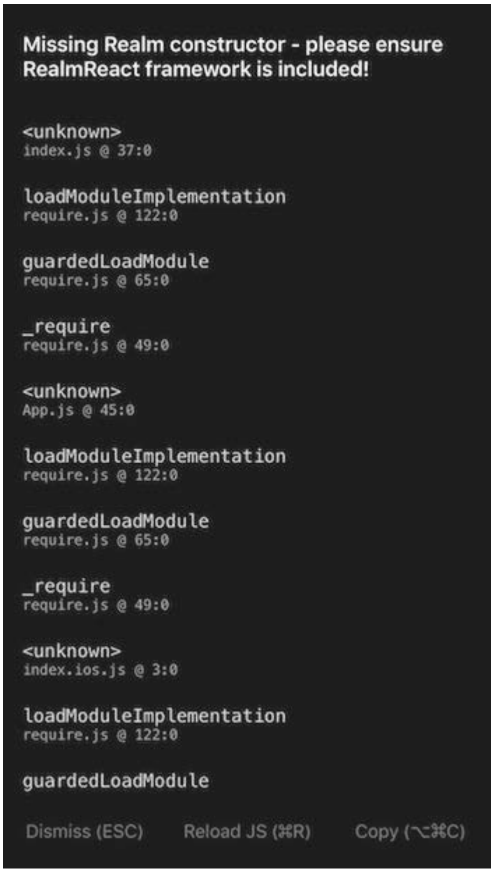
  it('calls componentDidMount', () => {
```

```
    const wrapper = mount(<Foo />);
    expect(Foo.prototype.componentDidMount.calledOnce).to.equal(true);
  });

});
```

## 4.6.2　UI 测试

XCode 7提供了一种新的测试应用的方式：UI测试。它可以通过编程的方式（用Swift语言）模拟所有用户行为。UI测试过程中可以点击元素、滚动、轻扫、在文本框中输入数据，以及执行其他任何用户行为。这种方式可以很好地自动测试整个应用（不像单元测试只能测试单个组件），避免出现React Native的“红色警告界面”。当某个地方出错时，比如没有正确导入某些模块，或者XCode没有添加某些原生iOS React Native模块时，就会出现类似以下例子中的红色警告界面，因为开发者忘了在XCode工程中添加某个库。由于单元测试无法保证应用能够正常运行，需要实施UI测试来覆盖所有UI界面。



以下示例展示了我们的简单UI测试，按以下步骤进行。

(1) 期望promo页面显示Sign In按钮。

(2) 注册测试用户，输入用户名（handle）、名字和姓氏。

(3) 前往消息界面。

(4) 前往我的账户界面。

(5) 滚动到底部，点击Logout（注销）按钮。

```
class ChatUITests: XCTestCase {

  override func setUp() {
    super.setUp()
    continueAfterFailure = false
    let app = XCUIApplication()
    app.launchEnvironment["TESTING"] = "1";
    app.launch()
  }

  func waitForElementAndTap(element: XCUIElement,
      timeout: NSTimeInterval = 50) {
    expectationForPredicate(NSPredicate(format: "exists == true"),
                            evaluatedWithObject: element, handler: nil)
    waitForExpectationsWithTimeout(timeout, handler: nil)
    XCTAssert(element.exists)
    element.tap()
  }

  func testSignIn() {
    let app = XCUIApplication()
    waitForElementAndTap(app.otherElements[" Sign In"], timeout: 500)
    let username = app.textFields["handle"]
    waitForElementAndTap(username)
    username.typeText("testUser1")

    let firstName = app.textFields["firstName"]
    waitForElementAndTap(firstName)
    firstName.typeText("John")

    let lastName = app.textFields["lastName"]
    waitForElementAndTap(lastName)
    lastName.typeText("Smith")

    waitForElementAndTap(app.otherElements[" Continue"])

    let rightNav = app.otherElements["rightNavButton"]
    waitForElementAndTap(rightNav)

    let messagesTitle = app.staticTexts["Messages"]
    waitForElementAndTap(messagesTitle)

    let leftNav = app.otherElements["leftNavButton"]
    waitForElementAndTap(leftNav)
```

4

```
    let profileBtn = app.otherElements["myAccountMenuItem"]
    waitForElementAndTap(profileBtn)

    let title = app.staticTexts["My Account"]
    waitForElementAndTap(title)

    let myAccount = app.otherElements["myAccount"]
    waitForElementAndTap(myAccount)

    let logout = app.otherElements[" Logout"]
    XCTAssert(logout.exists)
    myAccount.scrollUpUntilVisible(logout);
    logout.tap()
  }

}
```

这个测试很简单，因为它没有验证UI界面的内容，只验证了界面标题以及按钮是否出现。这就足够了，因为你可以通过单元测试来测试React Native组件。

UI测试应该不会影响生产环境数据库，但怎样用真正的应用来测试数据库呢？可以通过设置`launchEnvironment`属性来实现。

```
app.launchEnvironment["TESTING"] = "1"
```

然后在React Native代码内进行如下修改。

```
export default class App extends React.Component {
  constructor(props){
    super(props);
    settings.isTesting = props.TESTING != undefined;
  }
```

接下来React组件就会通过`settings.isTesting`来判断当前是否在进行UI测试。如果`setting.isTesting`为`true`，XMPP store就会连接测试环境服务器，而不是生产环境。

在React Native组件内，必须使用`testID`属性来设置专门的ID，供UI测试使用。另一种获取React Native Buttons组件的方式就是使用它们的文本值（Continue、Sign In、Logout），以空格开头。

### 4.6.3　快速更新应用

复杂的应用改动很频繁，而且甚至有了单元测试和UI测试后，仍然会出现致命bug。React Native允许你无需等待AppStore的审核，就可以远程发布应用的改动。很多组件都可用于完成这个目的，比如CodePush（https://microsoft.github.io/code-push/）或AppHub.io（https://apphub.io/）。

这种机制并不复杂，就是从服务端把JavaScript文件包复制到应用中，因此我们决定实现一个更灵活的解决方案。我们加入了打包签名、压缩还有数据加密以提供更好的安全性。

这个过程中最重要的部分在于恰当地管理应用版本。如果你添加了某些iOS原生组件，或者升级到新的React Native版本，那么你就不能为旧版应用发布JavaScript文件包。

### 4.6.4　版本控制系统

使用版本控制系统是合作开发成功的必要条件。我们使用Git，并把Github.com作为Git托管服务。

### 4.6.5　持续部署

在当前Git仓库的主分支上始终保存稳定且测试过的应用非常重要。有了持续部署系统，找到导致应用崩溃的代码就变得很简单（当某些自动化测试失败时）。并且这类系统不仅仅能够运行测试，还可以编译应用并将应用部署到iTunesConnect Testflight服务。我们使用Bitrise.io（http://bitrise.io/）作为部署服务器。

## 4.7　总结

我们希望通过本章对采用React Native开发TinyRobot应用的讨论，可以帮助你在开发自己的应用时有一个清晰的思路。

# 第 5 章 示例应用：Fixt

如果还没坏，就别急着修复它。

管理移动开发领域十分困难，因此有了React Native。编程语言有很多种：Objective-C、Swift、Java、C#，还有Visual Basic。未来几年内这个名单上还会出现更多的语言，可是谁能知道未来是什么样的呢？有些人可能会问，为什么要用另一门语言？我真的也要用JavaScript来开发移动应用吗？

之前解释过，React Native为开发团队带来了显著的优势。使用React Native能够联合你的代码库，这样就能很方便地进行跨平台测试、维护和改进应用。React Native还有一致性这一特点。用React Native开发的应用可以为不同平台的用户提供一致的体验。

React Native统一了视图与逻辑代码的主要部分，让我们可以专注思考故障分析策略，让应用变得独特而有创意。React Native和React之间的紧密关系还能够让我们在Web应用和原生应用之间维护共同的范式（Redux和无状态组件）。虽然Web应用和原生应用之间并未共享主要的视图代码，但无需切换编程风格能让我们保持专注。

## 5.1 何为 Fixt

Fixt是一项维修平板电脑和手机的贵宾服务。我们的产品及服务对普通消费者免费，只对企业用户收费。你可以从https://d.fixt.co/下载我们的iOS或Android移动应用。我们提供的平台可以让用户报告设备的损坏情况，并尽快对故障进行维修。通常当天就可以维修好设备并送还给用户。我们甚至可以让维修技术人员前往机场，在航班之间维修空中乘务员的手机！

我们的应用有一项故障分析的功能，利用React Native的原生层访问设备的传感器，当故障发生时检测到它。接着根据这一实时信息立刻采取措施，确保用户能得到快速方便的维修服务。我们利用与全世界数千个维修店的合作关系来帮助用户，为他们找到当地最具性价比的选择。我们与维修店协调以确保他们能提供客户设备所需的零件，还要保证他们能按客户的日程安排完成维修。从快速响应措施到保证性价比的各个方面，我们希望尽可能给用户最佳的维修体验。

选择React Native的原因在于，它允许我们为iOS和Android平台的客户提供无缝的体验。它还能让我们专注于特有的问题集合，而不用管那些用Objective-C和Java所开发的系统的复杂之处。我们的代码库之间共享UI代码，只有故障分析和安全过程需要为每个移动平台定制。对小型团队来说这是必不可少的，由于节省了时间和精力，才能够维护兼容两个平台的应用以及一些内部产品。

## 5.2 故障分析程序

> 人们认为计算机可以防止他们犯错，他们错了。有了计算机，你犯错会更快。
>
> ——Adam Osborne

### 5.2.1 快速分析与急救

Fixt的目的之一就是设备在使用周期内出现问题时就能被发现。假设你不小心摔了一下手机，某个内部元件损坏了，你什么时候能发现？或许要等到某天，你不能安全连接WiFi以接收重要邮件，或者错过潜在雇主的电话，你才会意识到手机坏了。我们希望防止这样的事发生在你身上。为此我们先要识别你的设备，以便根据它的型号来为你提供定制的体验。基本的故障检测之后，还需要深入检查设备更具体的特性，为你带来完全定制化的体验。

### 5.2.2 Platfom

React Native提供了一些工具用来识别所运行设备的常见特性。`Platform`工具包含了一些非常基本的平台信息。React Native文档中相应的部分解释了它一般用于设置条件逻辑，以便实现专门在iOS或Android平台上运行的代码。

```
import { Platform } from 'react-native';

// Android平台
Platform.OS // "android"
Platform.Version // 返回21或19
// iOS平台
Platform.OS // "ios"
Platform.Version // 未定义
```

`Platform`提供的属性对极其基础的分析来说够用了，但大部分情况下它无法帮你解决更深入的问题。操作系统信息最适用于一般的情况。我们用它定制任何与用户之间的自动交互过程。

```
render() {
  if (Platform.OS === 'ios') {
    return ( <IosSpecificComponent/> );
  }
```

5

```
    return ( <AndroidSpecificComponent/> );
  }
```

Android的版本号也很有用，可以在某些原生API不可用的情况下修改应用行为。这种用法在React Native自身的代码库中并不多，不过在自定义的原生模块中很常见，很值得关注一下。特别需要指出的是，发布到Play Store的应用可能包含错误，这会让特定API版本的用户完全无法使用你的应用。Android应用的发布过程没那么严格，因此你需要在宣称支持的所有Android版本上进行测试，确保不会在未包含某些后续版本原生API的设备上使用这些API。

```
render() {
  if (Platform.OS === 'android' && Platform.Version < 19) {
    return (
      <Text>
        This is returned for any Android version below KitKat
      </Text>
    );
  }

  return ( <Text> Otherwise this is returned </Text> );
}
```

Fixt团队分析设备问题或者修复bug时，需要操作系统和Android版本以外的更多信息。所幸React Native提供了另一个有用的模块，`NetInfo`。

### 5.2.3　NetInfo

`NetInfo` API通过提供详细的设备网络状态，让你能够获取关于设备环境的更多信息。`NetInfo`在iOS和Android上提供的功能与信息，详细程度不一致，因此接下来将分别对两个平台的情况进行介绍。

iOS上`NetInfo`提供的信息不太多。尽管iOS要比Android更早开发出这个API，它一直都只有很基础的功能。你能做的就是从很高的层面上获取当前网络状态，或者订阅事件监听器来检测网络状态的任何变化。

```
import { NetInfo } from 'react-native';

NetInfo.fetch().done((netState) => console.log(netState));

NetInfo.addEventListener('change', (netState) => console.log(netState));
```

这两个方法都返回数组`['none', 'wifi', 'cell', 'unknown']`中的一个值。重要的是，记得注销事件监听器。如果你忘了这样做，就会导致用户的电量白白浪费，这样他们就可能关闭或者卸载你的应用。

Android平台的`NetInfo`实现提供了更细粒度的网络信息。它与iOS平台的实现提供一样的方法，但返回更大范围的状态值（`NONE`、`BLUETOOTH`、`DUMMY`、`ETHERNET`、`MOBILE`、`MOBILE_DUN`、`MOBILE_HIPRI`、`MOBILE_MMS`、`MOBILE_SUPL`、`VPN`、`WIFI`、`WIMAX`、`UNKNOWN`）。这些值提供了足够的

细节来有效判断用户使用应用的场景，以便专注解决可能出现的特殊网络连接问题。

注意：React Native在Android上对`NetInfo`的实现方式没有提供所有可能的网络类型，甚至可能你认为很活跃的某种状态也获取不到。我们在尝试检查用户设备是否处于VPN网络时遇到了这个问题。

我们使用一台连接WiFi的Android 5.0设备，同时连接另一个第三方服务应用启动的VPN，这时我们发现了一个矛盾。网络连接是经过VPN的，但获取当前网络状态时`NetInfo`的返回值却是`WIFI`。这个意料之外的行为缘于Android平台的一个已知问题（https://issuetracker.google.com/issues/37068179）。

```
private String getCurrentConnectionType() {
  try {
    NetworkInfo networkInfo = mConnectivityManager.getActiveNetworkInfo();
    if (networkInfo == null || !networkInfo.isConnected()) {
      return CONNECTION_TYPE_NONE;
    } else if (ConnectivityManager.isNetworkTypeValid(networkInfo.getType())) {
      return networkInfo.getTypeName().toUpperCase();
    } else {
      return CONNECTION_TYPE_UNKNOWN;
    }
  } catch (SecurityException e) {
    mNoNetworkPermission = true;
    return CONNECTION_TYPE_UNKNOWN;
  }
}
```

当网络传输经过WiFi下的VPN时，`NetInfo`返回`WIFI`，这是因为React Native的NetInfoModule.java文件中的`getCurrentConnectionType`方法用到了Android的`ConnectivityManager`来获取活跃的网络信息。设备既连接了WiFi又使用了VPN的情况下，用`getActiveNetworkInfo`查询设备，`ConnectivityManager`的返回值为WiFi连接。为了绕过这个问题，我们实现了一个原生Android方法`isVpnActive`来真正地检测VPN连接。

```
@ReactMethod
public void isVpnActive(Promise promise) {
  Network[] networks = mConnectivityManager.getAllNetworks();

  for (int i = 0; i < networks.length; i++) {
    NetworkCapabilities capabilities
      = mConnectivityManager.getNetworkCapabilities(networks[i]);

    if (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_VPN)) {
      promise.resolve(true);
      return;
    }
  }

  promise.resolve(false);
}
```

这个方法利用ConnectivityManager获取设备所处的所有网络。接着检测这些网络，看它们是否拥有网络能力TRANSPORT_VPN。如果某个网络有这个能力，那就说明VPN正在该设备上运行，方法返回true。反之，在检查每个网络后返回false。

遗憾的是，这样无法保证你的应用，或者任何其他应用，在这种情况下真的在使用VPN连接。很可能VPN只是用来管理某些特殊应用的网络传输。如果你靠isVpnActive方法来确认应用正通过VPN进行连接，可能会得到错误的结果。

针对这种情况，更好的解决方案是利用应用内的路由追踪功能，再次确认VPN的使用情况。这样做比起在原生代码内检查设备网络要慢一些，因为你会遇到请求延迟，尤其是如果用户处于糟糕的网络环境中，延迟会很久。因此，更可取的做法是在应用需要结果之前完成这类请求。或者可以采取另一种方式，Netflix似乎使用并维护着一个已知的VPN IP地址列表（https://www.howtogeek.com/239616/how-to-watch-netflix-hulu-and-more-through-a-vpn-without-being-blocked/）。当用户从这些IP地址向你发送数据时，你就可以确定他们正在使用VPN。

## 5.2.4　Fixt 的设备参数模块

我们在Fixt应用中通过调用React Native以外的原生设备标识模块事先对设备进行识别，然后把标识信息作为常量提供给JavaScript层，这样应用就能访问设备信息，并且无需担心调用原生层方法耗费的时间。我们的团队非常认真谨慎，希望尽可能为用户带来流畅的应用体验。原生常量帮助我们实现了这个目标，因为访问它们比起异步调用原生层方法快了很多。当确实需要进行异步调用时，我们保证在需要之前完成调用，这样用户就不用等待信息返回了。

我们开发了一个开源模块，提供应用所运行设备的详细信息。react-native-device-specs这个包的用途相当直截了当，可以为你提供三个关键的信息：运营商、存储空间，以及设备型号/平台。

首先，该包提供设备平台代码。这是唯一标识设备的一个字符串。你可以在Google Play支持网站（https://support.google.com/googleplay/answer/1727131?hl=en-GB）上找到Android设备的平台列表，以及在iPhone Wiki（https://www.theiphonewiki.com/wiki/Models）上找到iOS设备的标识符列表。通过这个代码，你就可以判断用户使用哪个型号的设备访问你的应用。这种绝佳的方式能够收集那些影响部分特定用户的特殊bug。你还可以利用这个信息来弄清你的用户最普遍使用哪些设备，这对于测试新的应用布局极有帮助。

React Native设备参数模块还提供了运营商信息。设备没有SIM卡或者在模拟器中的情况下，返回字符串'No Carrier'。你可以通过运营商信息大致了解你的用户来自世界何处。这项信息还可以用来诊断应用崩溃的起因，如果错误影响特定的运营商或者那些没有SIM卡的用户，那么运营商信息就必不可少了。

最后，该包可以让你访问设备实际的存储空间。这与广告宣传的存储空间不一样。这项信息

不是一个整数，而是设备上可用存储区块的实际数量。它让你在更细的粒度上对用户设备有所了解。你可以利用这项信息进行目的明确的广告活动或者下载建议信息，并针对设备存储空间比较特殊的用户提供个性化体验。

我们在Fixt应用中利用这些信息来检测特定设备型号的问题。某些情况下，这可以帮用户少去一次维修店。型号信息也让我们了解到特定问题影响某个设备的频率。如果用户的设备无法维修，运营商和存储空间信息有助于更换合适的设备。通过编程的方式收集这些信息，我们让设备维修变得像点一个按钮那么简单。

### 1. 实现

该包的JavaScript层代码非常直观，就不再介绍了。下面将简要概述Objective-C和Java的代码，让你了解如何扩展与修改这类包。

### 2. Objective-C

从一个很简单的头文件开始（RNDeviceSpecs.h）。

```
// 导入RCTBridgeModule
#import "RCTBridgeModule.h"

// 设置界面
@interface RNDeviceSpecs : NSObject <RCTBridgeModule>

@end
```

接着在RNDeviceSpecs.m文件中通过以下代码创建所需的一切。

```
// 导入头文件
#import "RNDeviceSpecs.h"
// 包含sys类，用于访问系统
#include <sys/types.h>
#include <sys/sysctl.h>

// 还需要CoreTelephony类
@import CoreTelephony;

// 要实现的内容
@implementation RNDeviceSpecs

// 要导出的React Native模块
RCT_EXPORT_MODULE()

// 方法写在此处

@end
```

写在RCT_EXPORT_MODULE中的第一个方法用来获取设备型号。

```
- (NSString *)platform {
    size_t size;
    sysctlbyname("hw.machine", NULL, &size, NULL, 0);
```

```
    char *machine = malloc(size);
    sysctlbyname("hw.machine", machine, &size, NULL, 0);

    NSString *platform = [NSString stringWithUTF8String:machine];
    free(machine);
    return platform;
}
```

该方法以字符串形式返回平台名称。

头两行代码通过sysctlbyname方法访问"hw.machine"，返回完整的缓存空间。sysctlbyname方法通过字符串来读写系统信息。接下来的两行代码为size变量分配数个字节的内存，并用它来保存访问"hw.machine"时返回的完整缓存。最后，把这项信息转换成字符串，释放内存并返回设备平台（型号）。

下一个方法用来获取设备的运营商信息。

```
- (NSString *) carrier {
    CTTelephonyNetworkInfo *netinfo = [[CTTelephonyNetworkInfo alloc] init];
    CTCarrier *carrier = [netinfo subscriberCellularProvider];
    NSString *carrierName = carrier.carrierName;
    if (carrierName == nil) {
        carrierName = @"No Carrier";
    }

    return carrierName;
}
```

该方法以字符串形式返回设备的运营商名称。开始先初始化CTTelephonyNetworkInfo类的实例。接着用这个实例通过subscriberCellularProvider方法获取CTCarrier对象。从这个对象中就可以直接获取到carrierName并返回它。唯一的问题在于运营商名称可能不存在（平板或者无可用SIM卡的手机会有这种情况），这种情况下我们把carrierName设置成"No Carrier"。

最后一个方法用来获取设备的存储空间大小。我们使用NSSearchPathForDirectoriesInDomains获取设备的目录路径，接着通过NSFileManager获取这些路径的属性。这样就可以很容易地取得以字节为单位的文件系统大小，再进行一些基本计算就可以把单位转换成千兆。这样可以很好地大致估算广告所宣传的存储空间。后面会稍微介绍怎样计算这个值。

```
- (NSInteger) totalDiskSpace {
    NSNumber *fileSystemSizeInBytes = 0;
    NSInteger fileSystemSizeInGBytes = 0;
    NSError *error = nil;
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSDictionary *dictionary = [[NSFileManager defaultManager] attributesOf-
FileSystemForPath:[paths lastObject] error: &error];

    if (dictionary) {
        fileSystemSizeInBytes = [dictionary objectForKey: NSFileSystemSize];
```

```
        fileSystemSizeInGBytes = [@([fileSystemSizeInBytes floatValue] / 1E
+9) integerValue];
    } else {
        NSLog(@"Error Obtaining Disk Memory: Domain = %@, Code = %@", [error
domain], [error code]);
    }

    return fileSystemSizeInGBytes;
}
```

最后，封装一个方法用来调用上述的所有方法，然后返回一个常量对象。

```
- (NSDictionary *)constantsToExport
{
  return @{@"platform": [self platform], @"carrier": [self carrier], @"disk-
Space": [NSNumber numberWithInt:[self totalDiskSpace]]};
}
```

如你所见，访问基本的原生iOS设备信息一般来说很简单。通过获取用户设备上的此类信息，就能更方便地为用户提供一致的体验。React Native所提供的设备信息对于定制JavaScript代码来说足够了，但是定位那些影响一小部分用户的错误时，具体设备环境的附加信息就显得至关重要了。

如果你使用Redux并且追踪action记录，拥有设备信息以及系统版本号就可以非常容易地复现用户的操作流程。据我所知，我的团队曾遇到一些非常特殊的问题，特别是发布新的软件更新时。拥有用户设备的参数可以让我们复制用户的环境。这样就十分便于分析并修复这些问题。

希望你已经可以独立解决这些问题，并开始自己查阅iOS文档。文档中有大量非常特殊的方法，访问它们是一种在现实场景中监控应用的绝佳方式，这样就能确保用户在每台设备上都能享受到一致、流畅和原生的体验。

### 3. Android

该包的Android部分和iOS部分有一样的方法。不管你用的是Android还是iOS平台，我们都努力让JavaScript层能够很方便地访问这个包。关于创建文件来扩展ReactPackage类的方式没有什么好介绍的，因此接下来就直接跳到该模块更独特的代码部分。

```
@Override
public Map<String, Object> getConstants() {
  final Map<String, Object> constants = new HashMap<>();

  String platform = Build.MODEL;
  constants.put("platform", platform);

  ReactContext reactContext = (ReactContext)getReactApplicationContext();
  TelephonyManager manager = (TelephonyManager)reactContext.getSystemService
  (Context.TELEPHONY_SERVICE);
  String carrier = manager.getNetworkOperatorName();
  constants.put("carrier", carrier);

  String diskSpace = bytesToHuman(totalMemory());
```

```
    constants.put("diskSpace", diskSpace);
    return constants;
  }
```

如你所见，Android平台上访问设备参数也很简单。头几行代码初始化了常量Map，然后从Build类中获取设备型号。接着取到React应用的执行环境，用它访问TelephonyManager类的实例。通过该实例取得运营商名称。最后用了几个私有方法。这里将着重介绍最有趣的totalMemory方法。如果你对bytesToHuman方法很感兴趣，可以在Fixt的react-native-device-specs仓库中查看它的源码。

```
private long totalMemory(){
  StatFs statFs = new StatFs(Environment.getExternalStorageDirectory().getPath());
  long total = (statFs.getBlockCountLong() * statFs.getBlockSizeLong());
  return total;
}
```

这里用到了Android代码库里的Environment类，它可以让你访问媒体状态常量以及不同的目录。我们访问外部存储目录，计算它的区块大小和数量。接着把这两个数字相乘得到整个存储空间大小。这并不是广告宣传里的存储大小，不过也很接近了。在代码库里这个方法可以用来获取广告宣传的存储空间，不过大部分情况下，实际存储大小就能满足需求了。

```
const storageSizes = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512];
const closestSize = (arr, goal) => {
  return arr.find((val) => {
    return val > goal ? val : false;
  });
};

const advertisedDiskSize = closestSize(storageSizes, actualDiskSize);
```

存储空间大小往往是2的乘方，然而也有例外情况。这种情况可以采用专门的逻辑来筛选出存储空间大小比较特殊的设备。为了简单起见，这里不再介绍。closestSize函数接受一个数组和一个目标大小作为参数，并找到数组中第一个大于目标大小的数。这就是广告宣传的存储空间大小。这种做法是有道理的，因为我们知道广告宣传的存储空间不会小于实际的存储空间，而且也不会比它大很多。因此，第一个大于实际存储空间的数，就是广告宣传的存储空间大小。

### 5.2.5　React Native 的统一思想

移动设备领域十分庞大且多样化。React Native试图统一这个领域，以便为用户提供一致的体验，同时也为开发者提供简单直观的开发环境。尽管如此，异常的情况总会出现。bug总是不可避免地混到代码库中。而且越特殊的bug越难以分析与解决。另外，有些bug还只在生产环境中出现，这种情况下诸如网络连接类型、特殊的设备型号以及操作系统版本等细节可能无法获知。

幸运的是，有大量工具可以追踪生产环境的bug，让这些未知因素为我们所了解。React Native提供了Platform和NetInfo模块。Fixt团队开发了react-native-device-specs。这些工具提供了基

础水平的设备信息，与第三方的或者内部的后端服务集成后有助于识别特殊的bug起因。Fixt团队使用了Mixpanel来聚合用户设备的统计数据，应用中使用了Crashlytics检测崩溃情况。通过这两个工具的结合，我们就能捕获严重的bug。在每台设备上收集的数据能够让我们复制用户的环境，了解他们遇到的特殊问题并立即推送修复方案。

## 5.3 身份验证

> 你曾经习惯给我的手机打电话。
>
> ——Drake

### 5.3.1 何为 Digits

Digits是Twitter提供的手机号授权（auth）服务，允许用户通过短信或者电话接收并输入一个验证码，以此验证自己的身份。用户输入验证码之后，应用会获得一个Twitter ID，可以用它与Twitter的API确认用户身份的真实性。Digits与其他身份验证策略（比如社交账号登录，以及传统的用户名密码组合）不一样，因为它从根本上保证你获取到的是一份真实且唯一的用户标识信息。

在应用中使用我们所实现的React Native Digits的主要好处在于，你能够接入并使用原生的身份验证方案，提供了用户手机号就保证你拥有他们的联系方式。此外，与传统的身份验证系统相比，Digits还有更多好处。垃圾邮件制造者可以利用注册机器人，在那些没有基于电话号码的服务（比如邮件密码式或者社交登录系统）上轻松创建数千个假账号。对他们来说，获取数千个手机号有很大难度，这样某种程度上可以保证你的用户是真实的人。

可以通过很多服务（如burner号码服务）获取临时的短信号码，不过这些服务要收费，因此垃圾邮件制造者使用它们的可能性不大。

我们的Digits实现方式相当直观，结合利用了静态和实例方法，让它的代码很清晰，我们很希望这个包能够被其他人用到。

简单起见，假设你已经通过`npm install --save react-native-digits`从npm上下载了这个包，并按照使用说明把它连接到你的iOS与Android代码中。

我们实现的Digits只需很少的配置。你需要添加Digits的API密钥，另外针对Android平台还要在styles.xml中添加一些信息。

### 5.3.2 在代码内集成 Digits

React Native Digits的用法很简单。我们希望你把它看成一个视图元素，而不是一个身份验证操作或者外部应用。要把Digits集成到你的代码内，只需简单地导入它，放到`render`方法中，并配置一些属性即可。

5

```
import Digits from 'react-native-digits';

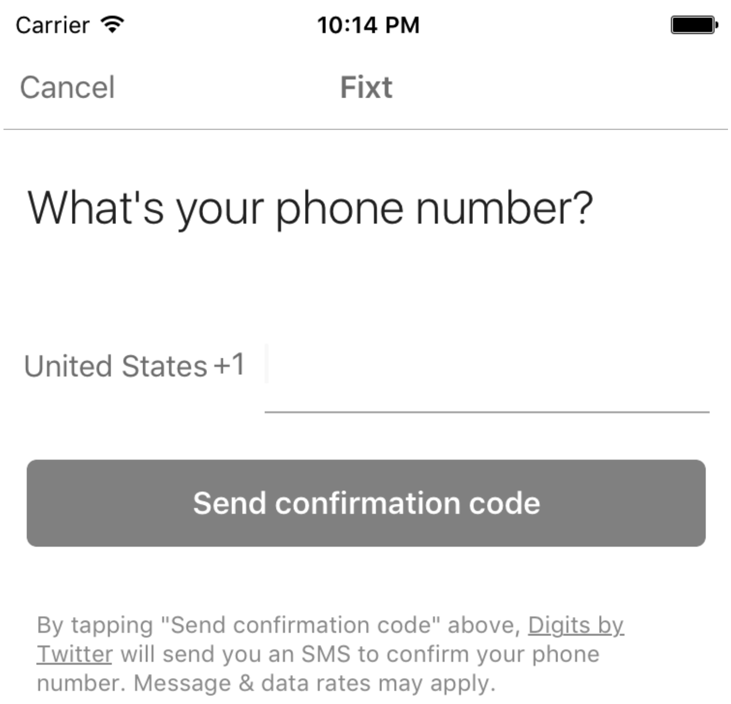
...

constructor(props) {
  super(props);
  this.state = { showDigits: false }
}

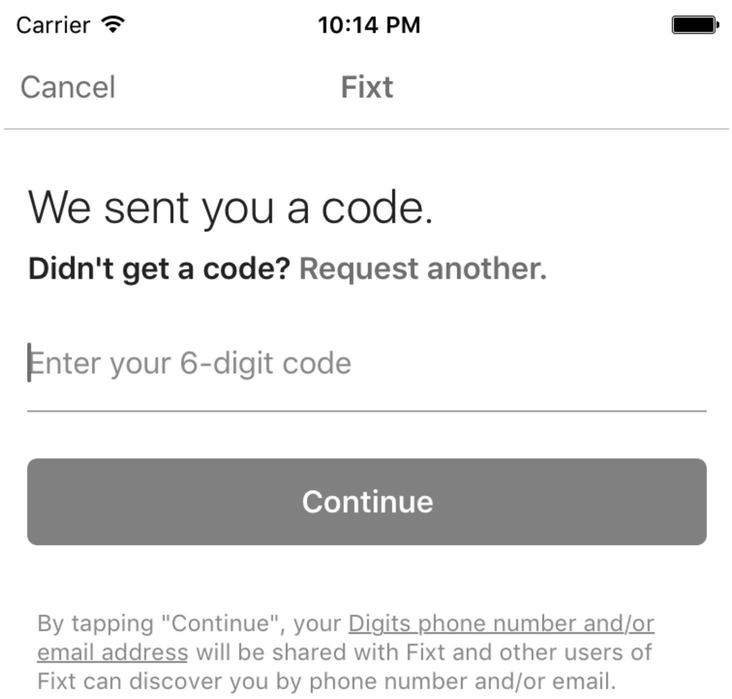
render() {
  return (
    <View>
      <Digits
        onLogin={ () => console.log('Logged in') }
        visible={ this.state.showDigits }
      />

      <Button
        onClick={ () => this.setState({ showDigits: true }) }
      />
    </View>
  );
}
```

点击按钮会把Digits组件从隐藏切换为可见，于是就能看到Digits的手机号输入提示。



用户输入手机号之后，就会收到一条短信，把里面的验证码输入到下一个界面的文本框中。如果几秒内没有收到验证码，界面上就会出现一个链接，可以让用户请求另一个验证码。第二个验证码将通过自动语音电话发送，因此从技术角度来说，Digits也对固定电话可用。不过从用户体验的角度来说，Digits主要用于短信身份验证。




用户输入Twitter发送给Digits的验证码之后，你会接收到Twitter发回的用户会话信息，通过这些信息你就能在服务端验证用户身份。最后一步非常重要。Digits不同于`auth0`那种完整的身份验证方案。收到的Twitter ID本身不能作为安全的身份验证方法。Twitter验证过用户所发送的Twitter ID为有效之后，你要给他们发送JWT令牌这样的信息。

## 5.3.3　样式

添加自定义样式非常简单。可以在JavaScript代码中指定样式，但不巧的是，在iOS平台上这样做只能改变Digits组件的颜色。

```
<Digits
  accentColor="(Color of buttons here)"
  backgroundColor="(Background color for all views"
/>
```

5

对于Android平台而言，需要在android/app/src/main/res/values/styles.xml文件中添加样式。这样做的原因是Digits在Android平台上使用主题来设置样式。不巧的是，Android的主题无法通过编程的方式进行更新，因此对原生Android代码中的主题，硬编码是唯一的选择。这意味着Android平台需要重新编译应用才能看到Digits的样式变化。如果你使用CodePush进行部署，那么还要在Play Store或者Amazon Store上提交更新，这样用户才能看到Digits主题的变化。大多数情况下这不是什么大问题，不过如果你重新设计了应用，并且提交了CodePush更新，最终Digits组件的主题样式可能与应用的其他地方有所差异。不过在Android平台上编译主题设置很简单。

把以下代码放到android/app/src/main/res/values/styles.xml文件中。一定要这样做，否则将导致错误发生。

```
<resources>

  ...

  <-- Customize this -->
  <style name="CustomDigitsTheme" parent="android:Theme.Holo.Light">
    <item name="android:textColorPrimary">@android:color/black</item>
    <item name="android:textColorSecondary">@android:color/darker_gray</item>
    <item name="android:windowBackground">@android:color/white</item>
    <item name="android:textColorLink">#000000</item>
    <item name="android:colorAccent">#000000</item>
    <item name="dgts__accentColor">#000000</item>
  </style>

</resources>
```

以上就是修改Digits样式的全部内容！

## 5.3.4　回调函数

我们提供了JavaScript层的两个回调函数，一个用于请求成功（`onLogin`），另一个用于请求失败（`onError`）。出现错误的情况下会把原生层抛出的错误传递给你。请求成功后会把已进行身份验证的用户的凭据传递给你。你需要把这些信息发送到自己的服务端，以便和Digits确认这些凭据信息的合法性。接着就可以很安全地把JWT令牌发送给用户，或者使用任何特定的身份验证方案。

React Native Digits的优秀之处在于它封装了原生的Digits实现。这意味着你无需关心Digits的内部状态，也不用维护Digits与Twitter服务器之间的安全连接。然而，这样也存在缺陷。Digits没有提供`onCancel`回调函数，因此无法把它提供给JavaScript层。用户取消Digits身份验证流程可能导致奇怪的bug，这取决于你在代码中使用React Native Digits的方式。

举例来说，以下代码中用户取消Digits身份验证流程后，会导致他不能再次使用Digits身份验证。

```
export default class MyApp extends Component {
  constructor(props) {
    super(props);

    this.state = { showDigits: false };
  }

  showDigits() {
    this.setState({ showDigits: true });
  }

  render() {
    return (
      <View>
        { showDigits ? (
            <Digits
              onLogin={ () => console.log('Logged in') }
              visible={ true }
            />
          ) : (
            <Button onClick={ () => this.showDigits }>
              Open Digits
            </Button>
          )
        }
      </View>
    );
  }
}
```

用户取消Digits身份验证后，组件内部状态没有更新成`showDigits: false`，因为缺少`onCancel`处理函数。可惜解决这个问题的方式不太优雅。我们把`showDigits`方法改写成以下形式。

```
showDigits() {
  this.setState({ showDigits: true });
  setTimeout(() => this.setState({ showDigits: false }), 1000);
}
```

任何平台上`setTimeout`回调函数都不会隐藏Digits组件，因为Digits独立于视图栈之外。它处于JavaScript层之上，不受该层的视图变化影响。把状态设置回`showDigits: false`可以让Android平台的用户取消身份验证后再次进行尝试。后面对于这个问题的另一种解决方案，将通过实现静态方法来启动Digits。但这样做也有代价，很难看清应用视图发生了什么。

### 5.3.5 注销

我们提供了静态方法用于注销用户。

```
static logout() {
  RNDigits.logout();
}
```

这段代码很好理解。你可以在JavaScript代码的任何地方通过静态方法来调用`Digits.logout()`。如果通过`prop`属性来控制它，需要把组件放到你希望用户进行注销的视图中，而这并不是该方法的意义所在，因为它不像Digits身份验证组件那样以视图为基础。原生层的实现很简单。

以下是Android的代码。

```
@ReactMethod
public void logout() {
  Digits.getSessionManager().clearActiveSession();
}
```

我们简单地使用Android的`SessionManager`来清除活跃的`DigitsSession`会话。

以下是iOS的代码。

```
RCT_EXPORT_METHOD(logout)
{
  [[Digits sharedInstance] logOut];
}
```

这两个平台的代码基本上没什么区别。

如果你想要实现自己对原生Digits注销操作的封装，可以利用这些方法开发一个注销按钮，并把它作为你的包或者UI组件的一部分，而不用静态方法的形式。

### 5.3.6　实现

以下是Fixt React Native Digits完整的JavaScript层实现。如你所见，它相当简单，而且做到了模块化。我已经添加了注释，以便详细解释每行代码。

```
export default class Digits extends Component {


  componentWillReceiveProps(props) {
    // 如果Digits原本不可见且现在需要显示
    if (props.visible && this.props.visible == false) {
      // 那么显示Digits
      this.show();
    }
  }

  show() {
    // 这些颜色值可以从JS层定制iOS的Digits实现
    const { accentColor, backgroundColor } = this.props;
    const config = { accentColor, backgroundColor };

    RNDigits.view(config, (err, session) => {
      // 出错时触发错误回调函数
      if (err) {
```

```
        this.props.onError(err);
      // 否则触发onLogin回调函数
      } else {
        this.props.onLogin(session);
      }
    });
  }

  render() {
    // Digits本质上作为一个独立的应用，因此不需要渲染
    return false;
  }
}


Digits.propTypes = {
  // 这些属性是可选的，不过大多数情况下都会用到
  accentColor: PropTypes.string,
  backgroundColor: PropTypes.string,
  onError: PropTypes.func,
  // 这些属性是必需的，因为和Digits的基本功能相关
  onLogin: PropTypes.func.isRequired,
  visible: PropTypes.bool.isRequired,
};

Digits.defaultProps = {
  onError: (err) => console.warn(err),
  visible: false,
};
```

我们对Digits的实现方式不是构建这个包的唯一方式。它只是适用于我们的应用需求和编程架构。如果你在寻找自行开发封装Digits的工具，那么Digits的SDK文档是绝佳的入门读物。如果你需要封装Digits的示例，我们完整的实现代码，包括原生的部分，也是很有用的灵感来源。

### 5.3.7 数据维护

Digits提供了一种绝妙的方式来确保用户的真实性，还可以避开垃圾邮件制造者。然而使用Digits验证用户时会遇到一些数据维护的问题。幸运的是，你从Twitter会话取回的Twitter ID没有任何问题。无论用户是否注册了Twitter或者把账户绑定了手机号，Twitter每次都对每个手机号返回相同的Twitter ID。

问题在于，随着时间的推移，手机号可能会易主——人们可能搬家、去世，还有启用新号码。更换手机号后，他们不会通知你，让你帮助他们把账户转移到当前手机号名下。即使他们通知了你，你恐怕也不想帮他们一个个地绑定新手机号。

解决这个问题的一种思路就是，让用户可以选择用账户名和密码来登录。从责任的立场上看，这个解决方案似乎不错。某人买到他人的旧手机号后也能取得对方的账户，这不属于你的责任范畴。遗憾的是，这种用户体验很糟糕。用户不能访问自己的账户会感到很沮丧，并且他们会因自

己的敏感数据暴露给其他人而生气。此外，你还要维护另一个无法解决最初问题的的身份验证系统。

更好的选择是让用户通过用户名密码注册，但是让他们在注册过程中用Digits验证自己的账户。这种方式下，用户拥有唯一且一致的方式来访问账户，而你又能得到Digits的好处。这仍然保证了用户的真实性，还能获得他们的手机号。

某些应用不需要用户的任何特殊信息。Digits也是这类应用绝佳的身份验证候选方案。如果不需要用户信息，那么用户通过手机号交易取得他人的身份标识也就无关紧要了。这种情况下Digits可以作为完整的身份验证方案。你的应用会有一个简单原生的用户身份验证方法，并且还能从保证用户真实性方面受益。

## 5.4　建议：如何管理快速变化的生态

React Native的发展速度很快。如此迅猛的发展带来了许多创新与进步，同样也带来了一些挑战。浏览一下React Native的生态圈就会很有感触，因为几个月前还有用的建议和解决方案，现在很可能已经不再适用。幸运的是，React Native的核心团队十分出色，积极响应Github上的issue，并且Stack Overflow上也有大量的社区开发者。以下几点提示曾经帮助我们的团队专注于代码，避免了在弄清React Native的复杂生态上浪费时间。

### 5.4.1　让应用保持最新

没有让应用与React Native的最新版本保持一致会很危险。再怎么强调这一点也不为过。让应用保持最新，能够避免最终决定升级时却受阻于几个月内的重大更改。React Native提供了工具来简化更新过程。

如果你安装了React Native CLI工具，只需要在项目根目录输入`react-native upgrade`即可。如果项目的全新分支不包含未提交的改动，这项操作会更简单。原因主要在于这条命令不能完美地升级。它所做的只是重写需要更新的文件，仿佛新建了一个项目。升级过程中你需要手动检查每个文件，并决定哪些要保留哪些要升级。这个过程稍微有些麻烦，不过为了与React Native代码库保持一致，值得这样做。

如果你决定不进行升级，而要停留在特定版本的React Native上，就要确保在package.json文件中锁死版本依赖。这样当其他包自然而然地随着新版的React Native升级时，就不会出错，否则将很难调试。当你决定升级React Native时，也要确保检查其他依赖支持高版本的React Native后再升级它们。

### 5.4.2　浏览文档

过去一年中，React Native的文档有了很大的改进。文档官网提供了很有价值的信息，比如创建新项目时如何配置React Native、JavaScript层的知识，以及如何开发自己的原生模块。也可

以按照发布版本浏览文档，这样当你决定延迟升级时，就不至于无法查看适合当前React Native版本的文档。

然而，React Native还有很大一部分缺少文档，或者文档仍在编辑，尤其是原生代码的部分。尽管大部分使用React Native代码库的开发者的关注重点不是原生部分，但开发自己的原生模块时必然需要浏览原生代码。幸运的是，虽然官网上没有文档，但是React Native的原生部分包含详细的注释。理解如何自行开发的最佳方式就是熟读几个较底层的原生模块代码。

### 5.4.3　何处以及如何寻求帮助

寻找React Native相关信息的方式数不胜数。如果你要找特定的原生模块，可以尝试npm或者js.coach。如果找到的模块不能完全满足你的需求，还可以提交拉取请求（pull request），对这些项目之一进行改进，或者以它们为灵感来自己实现一个。

如果你的问题比较常见，应该先查阅React Native的Github仓库和文档。另一个支持React Native（也包括许多其他框架和语言）的资料来源就是Stack Overflow。也可以尝试一下Discord，这是一个拥有广大React以及React Native社区的聊天应用。无需多言，只要你尽可能清晰详细地描述你的问题，以上任何方式都能获得有帮助的回复。如果你提出了一个问题又自己解决了它，请同样提交解决方案。这样肯定能帮到以后遇到相同问题的人。

希望本书能够帮助你增长经验和知识，让你能够自如地使用React Native开发出自己的应用。

# 版权声明

**所在的开发团队规模较小，但想要为iOS和Android两个平台开发应用？**

**早就听说过React Native的大名，却不清楚是否适合开发自己的应用？**

**研究无数代码之后，想要了解更多React Native在当今业界的实际使用情况？**

**如果你有以上困惑，那么本书不容错过！**

- 了解React Native部署过程与原生模块的使用
- 用JavaScript、Java和Objective-C创建自定义原生组件、异步调用、第三方库链接
- 自定义构建脚本的实现，以及如何在iOS、Android、Web应用间共享代码
- 无bug移动应用的维护
- 静态类型检查、依赖注入以及应用状态管理
- 如何从UI中分离业务逻辑，如何实现UI测试
- 如何利用React Native实现特定用途

Deconstructing React Native Apps

# React Native

**应用开发实例解析**

Bleeding Edge Press
Publishing at the speed of technology

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机 / 软件开发 / 前端开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-46714-0

ISBN 978-7-115-46714-0
定价：45.00元

# 看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks